UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea:

Progetto e Sviluppo dell'Applet "Garbage Collector" per Java Card

Relatore: **Prof.ssa Cinzia Bernardeschi**

Relatore: **Prof.ssa Nicoletta De Francesco**

Candidato: Alessandro Castrucci

Anno Accademico 2003-2004

Indice

Indice i
Elenco delle Figure e Tabelle iv
Sommario vi
1. Introduzione 1
2. Garbage Collection 3
2.1 Tempi e modi di intervento 4
2.2 Algoritmi di Garbage Collection 7
2.2.1 GC conservativi e GC precisi 7
2.2.2 Reference Counting 8
2.2.3 Algoritmi di Tracing 9
2.2.3.1 Mark and Sweep 9
2.2.3.2 Compacting 10
2.2.3.3 Copying 11
2.2.3.4 Generational 12
2.2.3.4.1 RIFERIMENTI INTRAGENERAZIONALI 13
2.2.3.5 Un esempio 14
3. Applet per Java Card 17
3.1 La larga diffusione delle Smart Card 17
3.2 Architettura delle Smart Card 18
3.3 Modalità e Protocollo di Comunicazione 20
3.4 L'avvento delle Java Card 22
3.5 Sottoinsieme del linguaggio Java 23
3.6 La Piattaforma Java Card 23
3.6.1 La Java Card Virtual Machine (JCVM) 24
3.6.2 Il Java Card Runtime Environment (JCRE) 24
3.6.3 Le Application Programming Interfaces (API) 26
3.7 Le Java Card Applet 27
3.8 Modello di memoria delle Java Card 29
3.9 Firewall e meccanismo di condivisione 32
4. Progetto realizzativo 35

4.1 L'ambito di collocamento 35
4.2 Le idee alla base 36
4.3 L'algoritmo implementato 38
4.3.1 Strutture dati necessarie 38
4.3.1.1 Esempio di Soluzione attuale 39
4.3.1.2 Soluzione Proposta 42
4.3.2 Funzionamento 45
4.3.3 Alcune Considerazioni 46
4.4 L'ambiente di sviluppo 47
5. Il Prototipo gcApplet 49
5.1 Il package GC 49
5.2 L'interfaccia gcInterface e manuale d'uso 50
5.2.1 Vincoli al programmatore 51
5.2.2 gcInterface 51
5.2.3 Manuale d'uso 52
5.2.3.1 Installazione del package e registrazione con la gcApplet 52
5.2.3.2 Nomi delle variabili e NULL_ID 53
5.2.3.3 Metodo newObject 53
5.2.3.4 I 'Figli' delle variabili 55
5.2.3.5 Metodo assignObject 57
5.2.3.6 Metodo startBlock ed endBlock 60
5.2.3.7 Metodo sysGC 61
5.3 La gcApplet: strutture dati e metodi 62
5.3.1 Classe Oggetto 63
5.3.2 Classe Riferimento 64
5.3.3 Classe groupContext 65
5.3.4 Classe Azione 67
5.3.5 Classe fileLog 67
5.4 Classe gcApplet 69
5.5 Modello riepilogativo 70
6. Un caso d'uso 72
6.1 Installazione dell'applet client01 e ottenimento dello SIO 72
6.2 Creazione di un Oggetto 75
6.3 Assegnamento di un Oggetto 77

6.4 Gestione di più applet 78
6.5 Esempio con JCWDE 81
7. Conclusioni 87
8. Bibliografia 88

Elenco delle Figure e Tabelle

Figura 2. 1 Frammentazione dello HEAP 3
Figura 2. 2 Garbage Collector Seriale 5
Figura 2. 3 CMS Garbage Collector 6
Figura 2. 4 Garbage Collector Parallelo 6
Figura 2. 5 Mark and Sweep 9
Figura 2. 6 Compattamento dello heap 10
Figura 2. 7 Promozione in un Collector Generazionale 12
Figura 2. 8 Collector Generazionale 14
Figura 2. 9 Area della nuova generazione 15
Figura 2. 10 Durante una collezione minore 15
Figura 2. 11 Situazione dopo una collezione minore 16
Figura 3. 1 Alcune famiglie di carte attualmente in commercio 17
Figura 3. 2 Struttura Smart Card 19
Figura 3. 3 Comunicazione tra host e Java Card 21
Tabella 3. 1 Command APDU 21
Tabella 3. 2 Response APDU 21
Tabella 3. 3 Funzionalità Java supportate o meno 23
Figura 3. 4 Conversione dei File Class e Java Card Installer 24
Figura 3. 5 Il Java Card Runtime Environment 25
Figura 3. 6 Diagramma di Stato di un'applet 29
Figura 3. 7 Memoria nelle Java Card 31
Figura 3. 8 Passi per la condivisione di un SIO 34
Figura 4. 1 La tabella delle Carte per la vecchia generazione 36
Figura 4. 2 Un Oggetto in memoria 39
Figura 4. 3 Oggetti nello heap 41
Figura 4. 4 Modello schematico della gcApplet 45
Figura 4. 5 Un Esempio di comunicazione con Apdutool 48
Figura 5. 1 Struttura del package GC 48
Figura 5. 2 Struttura UML di gcInterface 51
Figura 5. 3 Struttura UML della classe Oggetto 63

Figura 5. 4 Struttura UML della classe Riferimento 64
Figura 5. 5 Struttura UML della classe groupContext 65
Figura 5. 6 Struttura UML della classe Azione 67
Figura 5. 7 Struttura UML della classe fileLog 67
Figura 5. 8 Struttura UML della classe gcApplet 69
Figura 5. 9 Struttura UML del package GC 71
Figura 6. 1 Strutture dati dopo l'installazione dell'applet Client01 75
Figura 6. 2 Strutture dati dopo la chiamata alla newObject 76
Figura 6. 3 Strutture dati dopo a) la creazione di ALIAS 77
Figura 6. 3 b) l'assegnamento all'oggetto puntato da VETT 77
Figura 6. 4 Struttura dati dopo un esempio più complesso 79
Figura 6. 5 Struttura riepilogativa dopo l'eliminazione del riferimento VETTB 80
Figura 6. 6 L'esecuzione dell'emulatore JCWDE 81
Figura 6. 7 L'output del JCWDE 82
Tabella 6. 1 Codici di interpretazione dell'output 86

Sommario

La caratteristica peculiare del linguaggio Java consiste nella sua semplicità e portabilità. Le Java Card introducono tali vantaggi in un sistema embedded con capacità computazionali ma risorse limitate.

Un'altra caratteristica di Java è la gestione automatica della memoria ad opera del garbage collector, che svincola il programmatore dal compito di compattare, spostare e deallocare oggetti. Questa caratteristica è, però, assente nelle specifiche delle Java Card. Il programmatore di applet complesse deve far fronte alla gestione della memoria con proprie routine inserite nel codice.

In questa tesi è stato affrontato il problema della gestione automatica della memoria in Java Card. Sono stati studiati i diversi tipi di Garbage Collector presenti in letteratura ed un algoritmo di Garbage Collection per le Java Card è stato proposto. Tale algoritmo è di tipo non conservativo, incrementale ed è basato sul Reference Counting.

Un prototipo di Garbage Collector è stato implementato come una Java Card Applet che, interagendo con le altre applet, crea un modello della memoria dinamica. Il prototipo sviluppato utilizza una struttura dati che lo rende flessibile verso l'implementazione di algoritmi di tipo Tracing.

1. Introduzione

Le Java Card adottano tutti gli standard della normativa ISO 7816 per Smart Card in termini di dimensione, contatti, alimentazione elettrica e protocollo di comunicazione, giocando sul fatto che ormai le Smart Card sono un prodotto altamente diffuso e supportato.

Se le Smart Card aggiungevano alle Memory Card un microprocessore (con, a volte, un coprocessore), le Java Card adesso introducono un supporto al linguaggio Java sotto forma di una virtual machine che interpreta il bytecode, appositamente convertito in un formato particolare .CAP (converted applet). Tale codice, assieme ad informazioni per il linking, è caricato sulla carta per mezzo dell'*installer*.

La piattaforma Java Card supporta solo un sottoinsieme del linguaggio Java, che include funzionalità che sono ben adattabili alla scrittura di programmi per Smart Card o altre piccole periferiche e che mantiene le capacità orientate agli oggetti del linguaggio ad alto livello.

Uno degli aspetti chiave del successo delle Java Card sta nella sicurezza: la politica di sicurezza del linguaggio Java ben si adatta al contesto delle Smart Card dove già si trovano funzioni di autenticazione e basilari applicativi di firma digitale. Inoltre la sicurezza è garantita anche dalle modalità di accesso ai dati che le varie Applet sono costrette a rispettare. E' infatti compito di un *firewall* di sistema suddividere gli spazi di memoria assegnati alle diverse applet in diversi contesti. E' impossibile, per un'Applet, creare un puntatore ad una zona di memoria che non gli appartiene, ed ogni Applet deve gestire lo spazio di memoria assegnatole.

La memoria è costituita da un quantitativo di RAM (Random Access Memory) limitato su cui si appoggiano le operazioni, ma quella su cui operano i programmi è in maggior parte di tipo EEPROM (Electrically Eraseable Programmable Read Only Memory). Questo significa che il programmatore non può pensare di abusare nella creazione di oggetti, poiché, oltre al rischio di esaurire le risorse, deve far fronte a tempi molto più lunghi (circa trenta volte) che una scrittura nella EEPROM comporta rispetto ad una scrittura nella RAM e deve tener conto del tempo di vita della EEPROM.

In questo scenario risulta evidente come operare manualmente sulla gestione degli oggetti sia complesso ed evidenzia l'utilità del meccanismo di gestione dinamica della memoria, il *Garbage Collector*, che è presente nel linguaggio Java.

La *Garbage Collection* determina automaticamente quali oggetti in memoria il programma non sta più usando, e li "ricicla" per altri usi. E' anche nota come "reclamo automatico di memoria" [GCFAQ].

Pur appesantendo l'esecuzione di un programma questa automatizzazione porta a numerosi benefici, tra cui una più rapida programmazione e meno soggetta ad errori. Inoltre i moderni Garbage Collector, si inseriscono nei "tempi morti" dei programmi alleggerendo l'attesa dell'utente.

Esistono diversi tipi di Garbage Collector, ma le specifiche delle Java Card non ne prevedono uno nel suo runtime environment: lasciano ai produttori libera scelta di implementazione [JVM].

Il Garbage Collector proposto in questa tesi è basato sul Reference Counting, e può essere installato sulla carta come una normale Applet (sarà chiamata, nel seguito, *gcApplet*). Analizzando le operazioni di creazione e modifica di oggetti fatta dalle altre applet, questa *gcApplet* esegue l'algoritmo su un modello di memoria semplificato e registra tutte le azioni intraprese in un file di Log che crea sulla carta. Tale file di Log può essere notificato all'esterno quando richiesto.

La tesi si suddivide nei seguenti capitoli:

Il Capitolo 2 riporta una panoramica sui tipi di algoritmo di Garbage Collection esistenti in letteratura con considerazioni sulle loro prestazioni.

Il Capitolo 3 introduce l'architettura delle Java Card e lo sviluppo di applet. Seguono il protocollo di comunicazione, i componenti caratteristici di supporto alle Applet, il *firewall* e i metodi di *Object Sharing*.

Il Capitolo 4 riporta il progetto di un prototipo di Garbage Collector per Java Card basato sul Reference Counting. Vengono motivate le scelte implementative e viene illustrato l'ambiente di sviluppo.

Il Capitolo 5 descrive il package del prototipo (*gcApplet*) e i requisiti delle applet utenti.

Nel Capitolo 6 è mostrato un esempio di uso del Garbage Collector che illustra le funzionalità della *gcApplet*.

Infine il Capitolo 7 riporta le conclusioni e sviluppi futuri.

2. Garbage Collection

Storicamente la Garbage Collection è stata parte integrale di molti linguaggi di programmazione, incluso il Lisp, Smalltalk, Eiffel, Haskell, ML, Scheme e Modula-3, ed è stata usata fino dai primi anni '60.

Nel linguaggio Java lo heap memorizza tutti gli oggetti creati da un programma in esecuzione. Gli oggetti sono creati dall'operatore "new", e la memoria per i nuovi oggetti è allocata nello heap a tempo di esecuzione. L'unico collegamento che rende fruibile un'area di memoria allocata, è il riferimento che punta all'oggetto allocato in memoria.

La Garbage Collection è il processo di deallocazione automatica di quegli oggetti in memoria che non sono più riferiti dal programma. Questo libera il programmatore dal doversi tener traccia di quando liberare la memoria allocata, allungando così i tempi di programmazione e aumentando le probabilità di bug nel codice. Si ha così una programmazione più produttiva e si assicura l'integrità dei programmi: non ci saranno crash della JVM dovuti ad una gestione erronea, accidentale o volontaria che sia, della memoria.

La Garbage collection può essere anche una valida difesa alla frammentazione dello heap, che accade durante l'esecuzione di un programma quando si liberano spazi di memoria non contigui e di dimensione variabile tali da non soddisfare eventuali richieste di nuove allocazioni. Si vengono così a formare dei "buchi" di spazio inutilizzato, come illustrato semplicemente in figura 2.1, dove un nuovo oggetto, pur essendo soddisfacente la quantità di memoria libera, risulta non allocabile.

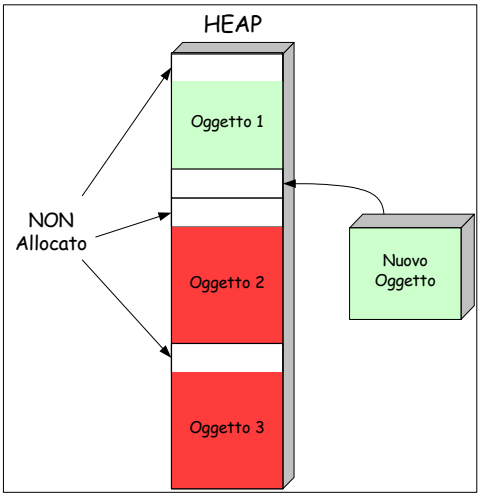

Figura 2. 1 Frammentazione dello HEAP

Non esiste alcun Garbage Collector ufficiale in Java; le specifiche della Java Virtual Machine asseriscono che lo heap deve essere soggetto a Garbage Collection ma non spiegano come [JVM]. Sta al progettista della particolare JVM a decidere come implementare tale heap.

2.1 Tempi e modi di intervento

Prima di fare una panoramica sui Garbage Collector esistenti attualmente, è interessante vedere come e quando essi interagiscono con il sistema e con il programma in esecuzione.

La reclamazione delle aree di memoria non più riferite, alla quale ogni Garbage Collector arriva secondo il proprio algoritmo, può accadere in tempi dipendenti dalle seguenti politiche:

- Chiamate esplicite
- Automaticamente

Nelle librerie Java, e più precisamente nel package 'java.lang' sono presenti due primitive, System.gc() e Runtime.gc(), grazie alle quali è possibile invocare esplicitamente il meccanismo di Garbage Collection implementato nella virtual machine [JVM].

Alternativamente è possibile che tale chiamata possa avvenire in maniera automatica direttamente dal sistema quando, ad esempio, si sta esaurendo la memoria, o ogni volta che è creato un nuovo oggetto, o periodicamente, o nei tempi morti, spesso fatti da molti cicli di clock, in cui, ad esempio, un programma attende l'immissione dei dati da tastiera, o attende una comunicazione con un processo remoto.

Effettuare chiamate esplicite potrebbe essere utile se un programmatore sapesse a priori quando, nel proprio programma, si ha una necessità di allocare molta memoria oppure quando sapesse di aver dereferenziato molti oggetti e volesse "riciclare" tali spazi.

Purtroppo non è sempre possibile prevedere questi eventi, e neppure si può sempre sapere quando staremo per incorrere in una eccezione di tipo *Out of Memory* (memoria esaurita). Per venire incontro a questa pseudo-aleatorietà alcuni interventi possono accadere quando ce n'è bisogno (o quando si pensa che ce ne sia), ovvero in

maniera automatica. Questo tipo di chiamate se da un lato solleva l'utente da chiamate esplicite, dall'altro può introdurre un notevole *overhead* di esecuzione.

"Costringere" la collezione di oggetti proprio in un momento critico solo perché è scattato il timer impostato (chiamate automatiche periodiche) può dare molto fastidio se non addirittura essere dannoso. Comunque, nel caso di sistemi real-time, si presuppone che la scelta dei tempi di collezione sia soggetta a schedulazione e venga affidata a task con basse priorità anziché resa semplicemente periodicizzata. Col passare del tempo, sono stati creati dei garbage collectors sempre più *smart* ("astuti") che intervengono in maniere più consone alle esigenze dei programmi.

I Garbage Collector possono infatti intervenire in diversi modi:

- Serialmente (anche noto come "ferma il mondo")
- Speculativamente
- Incrementalmente
- Concorrentemente
- Parallelamente

La maniera di intervento più semplice (e anche la prima realizzata storicamente) è quella seriale. Il flusso di istruzioni dell'applet in esecuzione è interrotto per far spazio a quello di garbage collection, come banalmente illustrato nella figura 2.2.

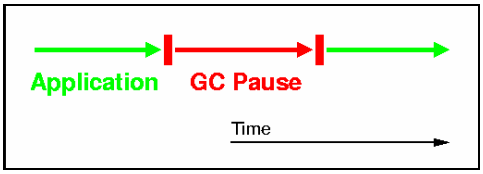

Figura 2. 2 Garbage Collector Seriale

Fermare completamente l'applicazione in esecuzione, anche se garantisce che alla ripresa si avrà il massimo dello spazio disponibile, può non essere la scelta ottimale. Proprio per questo si è pensato ad una modalità dove il Garbage Collector agisce solo quando l'Applicazione è inattiva (GC speculativo). Altrimenti la collezione può procedere a piccoli passi intervallati con la normale computazione (GC Incrementale). Quest'ultimo Garbage Collector fu introdotto dalla versione 1.2 di Java Developement Kit (JDK), riducendo le pause di Garbage Collection a spese del *throughput*, rendendolo adatto soprattutto in casi dove sia importante avere pause piccole (caso dei sistemi quasi real-time) [IBM].

Ulteriore evoluzione del GC incrementale è quello concorrente. Il JDK 1.4.1 ne implementa uno chiamato *Concurrent Mark&Sweep Collector* (CMS), di tipo generazionale, illustrato in figura 2.3 e il cui funzionamento sarà chiaro quando si parlerà di *Mark and Sweep* collectors nel Paragrafo 2.2.3.1. I GC concorrenti sono processi che girano in maniera concorrente con gli altri task del sistema. Nasce una casistica di possibili accessi e modifiche concorrenti ai dati che nell'incrementale non avveniva.

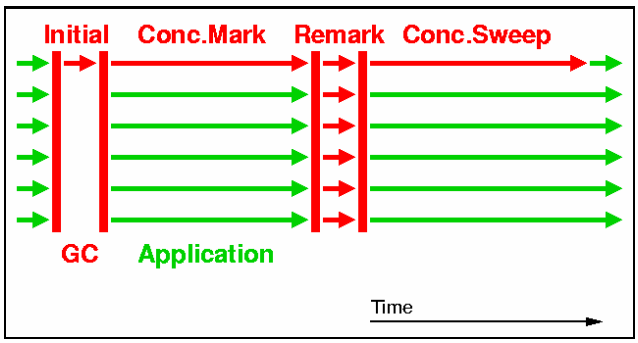

Figura 2. 3 CMS Garbage Collector

Le sue prestazioni dipendono dalle politiche di schedulazione esistenti nel particolare sistema operativo. Il tipo concorrente diventa parallelo se siamo in un contesto di CPU multiple, dove la Garbage Collection gira su uno o più processori contemporaneamente agli altri programmi. Risulta importante creare dei GC scalabili in modo da poter distribuire la complessità delle operazioni in più processi anziché uno solo. E' la release 1.4.1 del Java che ne prevede uno di tipo *Parallel Copying*, in figura 2.4 è illustrato il principio di disposizione temporale dei processi su ogni processore.

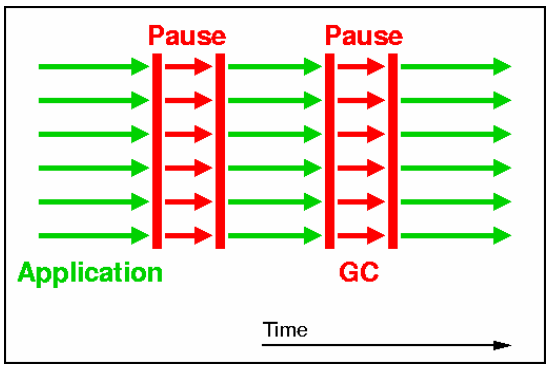

Figura 2. 4 Garbage Collector Parallelo

2.2 Algoritmi di Garbage Collection

Ogni algoritmo di Garbage Collection deve determinare anzitutto quali siano gli oggetti che costituiscono “garbage”, che costituisce la fase di *Garbage Detection*, dopodichè deve procedere alla fase di deallocazione tenendo conto della frammentazione dello heap.

2.2.1 GC conservativi e GC precisi

Si può subito fare una distinzione dei garbage collector circa l’accuratezza con cui essi interpretano gli oggetti in memoria.

Un Garbage Collector, infatti, esplora la memoria per individuare se contiene un riferimento ad un oggetto oppure ad un tipo primitivo (boolean, byte, char, short, int, long, float o double).

I tipi primitivi, infatti, sono allocati direttamente nello stack Java* o in una istanza, mentre gli altri sono dei sottotipi della classe *java.lang.Object* e sono allocati dinamicamente nello heap per mezzo dell’operatore new.

I Garbage Collector che decidano deliberatamente di non distinguere tra tipi primitivi o tipi riferimento si dicono **conservativi**. Nell’esplorazione della memoria, di fronte ad un dubbio se un oggetto sia un puntatore o meno, lo si interpreta come tale: questo porta ad una possibile permanenza di oggetti “garbage” in memoria. Di contro si ha un guadagno in termini di facilità di implementazione e velocità di algoritmo. Al contrario, se è possibile identificare in maniera certa se un dato valore in memoria è un puntatore o un suo simile, allora si è di fronte ad un Garbage Collector **preciso**.

Spesso si è disposti a pagare in termini di efficienza di algoritmo optando per un Garbage Collector conservativo. Non è detto che essi falliscano sempre e poi sono la scelta obbligata se si vuole adottarne l’uso in programmi che non siano stati scritti con l’idea di una gestione automatica della memoria. I GC conservativi sono adattabili

* ogni thread in esecuzione ne ha uno proprio

ad ogni sistema con poca o nessuna cooperazione da parte del compilatore o del linguaggio.

Uno degli svantaggi dei conservativi rispetto a quelli precisi sta nel fatto che essi non possono rilocare gli oggetti. Ne consegue una frammentazione della memoria con un conseguente degrado, a lungo termine, delle prestazioni.

Sono implementati in diversi linguaggi anche soluzioni **semi-conservative** o **parzialmente accurate**, che trattano alcune aree di memoria (stack o registri) in maniera conservativa ed altre (lo heap) in maniera precisa.

Una suddivisione iniziale di algoritmi è tra i **Reference Counting** ed i **Tracing** Collector.

2.2.2 Reference Counting

E' stata storicamente la prima tipologia di algoritmi introdotta. Si basa sull'associare ad ogni oggetto creato un contatore che conti il numero di riferimenti che riferiscono tale oggetto. Inizialmente posto pari ad uno, tale valore viene incrementato ogni volta che si aggiunge un riferimento all'oggetto. Viene decrementato ogni volta che il riferimento all'oggetto "muore" (esce di visibilità) o viene assegnato ad un altro oggetto. Se il contatore va a zero diventa eleggibile per la Garbage Collection. In occasione della deallocazione vengono decrementati anche eventuali altri contatori di oggetti da esso riferiti. In questa maniera una cancellazione può scatenarne delle altre a catena.

I vantaggi di questo algoritmo stanno nel fatto che non si interrompe il programma per troppo tempo (la Garbage Collection può accadere anche in piccoli passi), rendendolo particolarmente adatto ad ambienti real-time o comunque con requisiti temporali stretti. Gli svantaggi del Reference Counting sono che si aggiunge un *overhead* dovuto al fatto di dover memorizzare un contatore per ogni oggetto e doverci accedere ad ogni modifica del riferimento o uscita di *scope*. Altro problema è che il Reference Counting non individua i cicli, ovvero due o più riferimenti che si referenziano l'uno con l'altro, ad esempio una lista ciclica di oggetti: i contatori di tali oggetti non saranno mai nulli e la catena di eliminazione non avrà mai inizio.

2.2.3 Algoritmi di Tracing

Gli algoritmi di tracing si basano su una esplorazione della memoria a partire da un set di *radici* fino all'esaurimento dei collegamenti per verificare quali oggetti siano raggiungibili. Gli oggetti che non vengono toccati da tale tracciamento sono eleggibili per la Garbage Collection. La scelta delle radici è fatta considerando tutti i possibili oggetti che sono sempre raggiungibili dal programma in esecuzione, ciò comprende ogni riferimento presente nello stack di ogni thread, nei campi e nei metodi delle classi caricate.

2.2.3.1 Mark and Sweep

Esistono moltissimi tipi di Tracing Collector, uno dei primi ideati è stato il **mark and sweep** (marca e butta via), dove durante l'esplorazione dei riferimenti si marcano tutti gli oggetti che si incontrano (lo si può fare sia settando un bit che tenendosene una mappa a parte). Finita questa fase si scorrono tutti gli oggetti: quelli che non sono stati marcati verranno eliminati.

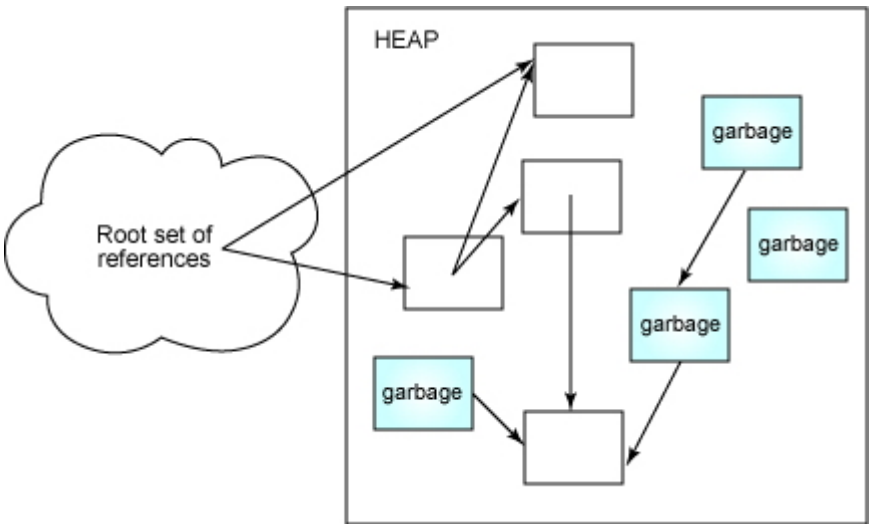

Figura 2. 5 Mark and Sweep

E' un algoritmo molto facile da implementare, inoltre settare un flag è molto meno dispendioso, in termini di memoria occupata, che non tenere un contatore. E' stato

inoltre dimostrato che l'efficienza degli algoritmi di tracing è proporzionale all'ammontare di memoria utilizzata. La fase di *marking*, però, è molto delicata e dovrebbe avvenire in modalità seriale, cosa che potrebbe richiedere un tempo di attesa lungo. La fase di *sweep*, anche se incrementale, deve percorrere tutta la memoria con notevoli dispendi di tempo e, una volta liberati gli oggetti "garbage", la lascia frammentata.

Il **CMS** Collector, introdotto nel Paragrafo 2.1, cerca di rendere concorrente l'operazione di marking facendo precedere una piccola pausa seriale dove sono identificati il set di oggetti immediatamente raggiungibile dalle radici. Dopo c'è la fase concorrente dove si prosegue nella scansione a partire da quel set. Alla fine c'è una ultima fase di *marking*, chiamata *remark*, che colma la lacuna dovuta al fatto che nella fase concorrente potrebbe essere stato modificato qualcosa, e finalizza tale fase garantendo che tutti gli oggetti saranno stati visitati. La fase di *sweeping* è anch'essa resa concorrente, tanto ormai gli oggetti garbage saranno già stati individuati.

2.2.3.2 Compacting

Per combattere la frammentazione dello heap, una delle soluzioni adottate anche nei primi *mark and sweep* collectors è stata la compattazione.

Al termine della fase di *sweep*, tutti gli oggetti riconosciuti come "vivi" dal Garbage Collector vengono spostati verso un estremo dello heap. Nella figura 2.6a quelli individuati essere "garbage" sono marcati con una 'X'. In questa maniera si ottiene una zona contigua di memoria libera (area tratteggiata della figura 2.6b) dove il programma può allocare più facilmente i nuovi oggetti. Tutti i riferimenti agli oggetti rilocati devono essere aggiornati affinché non perdano di consistenza.

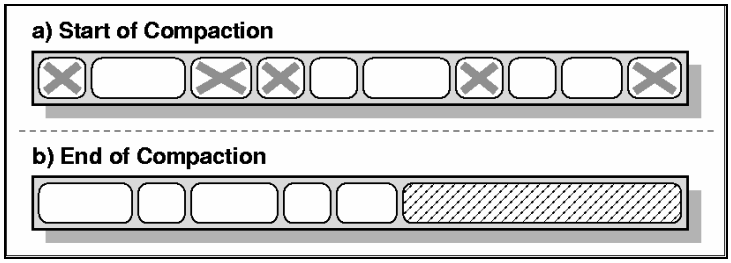

Figura 2. 6 Compattamento dello heap

Tale aggiornamento dei riferimenti è reso più semplice introducendo un ulteriore livello di indirezione: i riferimenti agli oggetti non indirizzeranno più direttamente lo heap ma andranno a riferire un gestore dell'oggetto (presente in una tabella dei gestori). A quel punto non si renderà più necessaria la rilocazione di tutti i riferimenti ma basterà aggiornare il solo gestore.

2.2.3.3 Copying

Basato sullo stesso principio di muovere gli oggetti per evitare la compattazione, la tecnica del *copying* ovvia alla necessità dei gestori di oggetti. Lo heap è suddiviso in due zone di memoria: una destinata all'allocazione degli oggetti e una di spazio libero.

Il programma, durante la sua esecuzione, alloca gli oggetti nello spazio riservato e lascia inviolato lo spazio libero. Quando l'area oggetti risulta piena (e probabilmente frammentata) si "ferma il mondo" e tutti gli oggetti non individuati essere "garbage" vengono copiati in maniera contigua nella zona di memoria libera. Alla fine i ruoli delle due aree vengono invertiti.

Tra questo genere di algoritmi il più semplice è lo **stop and copy**, e la complessità è proporzionalmente dipendente dal numero di oggetti "vivi" piuttosto che dallo spazio di memoria disponibile come invece era per il *mark and sweep*. Uno svantaggio è che gli oggetti più longevi saranno copiati di continuo ad ogni collezione, inoltre si renderà ancora necessario la rilocazione dei riferimenti nel nuovo spazio.

I Garbage Collectors di copia evitano la frammentazione al costo di una continua rilocazione degli oggetti. E' stata ideata una variante del *mark and sweep* che combina queste due tecniche e che si chiama **mark compact**. L'idea è quella che, durante la fase di *mark*, gli oggetti percorsi a partire dalle radici vengono subito copiati al fondo dello heap e alla fine quest'ultimo risulterà molto simile a quello lasciato da un algoritmo di copia, con la differenza che gli oggetti a lunga vita non saranno ricopiati ogni volta ma tenderanno ad accumularsi sul fondo dello heap.

Le prime release di JDK (fino alla 1.2), implementavano un Collector a thread singolo di tipo *mark and sweep* ed uno di tipo *mark compact*.

2.2.3.4 Generational

JDK 1.2 introduce un nuovo tipo di Garbage Collector che si basa sulle seguenti osservazioni, note come “ipotesi di debolezza generazionale”:

- La maggior parte degli oggetti che vengono creati in un programma sono destinati a “morire giovani” (spesso più del 95%!)
- Quegli oggetti che non muoiono subito sono molto probabilmente quelli a lungo termine, e sono molto pochi.

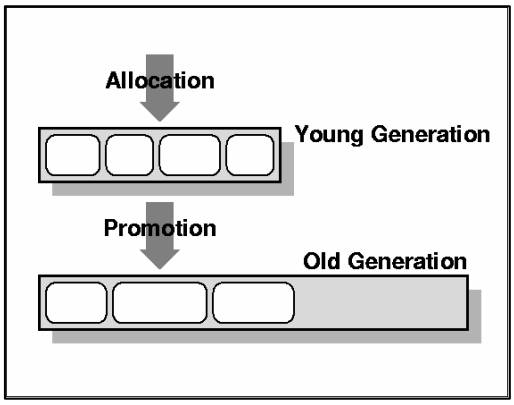

Figura 2. 7 Promozione in un Collector Generazionale

Come suggerisce il loro nome, questo tipo di algoritmi mantiene in memoria gli oggetti suddivisi in due o più generazioni ad ognuna delle quali ne è assegnata una porzione.

I nuovi oggetti creati vengono tutti messi con la generazione più giovane in maniera contigua (tale area di memoria è detta *eden*). Quando l’*eden* spazio risulta pieno si esplora, a partire dalle radici*, la sola generazione più giovane, per reclamare gli oggetti “morti”. Questa visita è nota come *collezione minore*. Quelli che sopravvivono ad una o più collezioni minori, a seconda delle scelte implementative, sono promosse alle generazioni più vecchie.

Periodicamente vengono fatte Garbage Collection anche nelle vecchie generazioni (queste sono note come *collezioni maggiori*). Dal momento che tale generazione è

* La radici o *root*-set sono presenti in quanto è sempre un algoritmo di tipo *tracing*.

confinata in uno spazio a sé stante, si potrebbe scegliere di utilizzare un diverso tipo di algoritmo per la collezione.

Le aree riservate alle due generazioni hanno diverse caratteristiche:

- **Nuova Generazione**. Ha un'area piccola ed è collezionata frequentemente. Dal momento che la maggior parte degli oggetti qui sono destinati a morire presto, il numero di quelli che sopravvive sarà esiguo e le collezioni saranno molto efficienti.
- **Vecchia generazione.** E' molto più grande l'area che occupa ma cresce con ritmi molto lenti. Qui le collezioni sono costose in termini di tempo, ma avvengono assai di rado. Per alleggerire il costo di una collezione maggiore spesso si suddivide questa generazione in più parti, ordinandole gerarchicamente e assegnandogli spazi a parte.

2.2.3.4.1 RIFERIMENTI INTRAGENERAZIONALI

Le collezioni minori non coinvolgono gli oggetti della vecchia generazione, i quali possono però avere dei riferimenti agli oggetti della nuova generazione che devono essere gestiti. Si risolve il problema aggiungendo tali riferimenti al set di radici.

Questi riferimenti intragenerazionali possono essere generati da una modifica di un oggetto della vecchia generazione che viene fatto puntare ad uno della nuova, oppure possono nascere dal fatto che di due nuovi oggetti che si riferiscono uno è promosso alla vecchia generazione. In entrambi i casi il garbage Collector deve tener conto di tali riferimenti. Potrebbe farlo esplorando tutta la vecchia generazione in cerca di puntatori alla nuova, ma occorrerebbe molto tempo. Altrimenti potrebbe, in fase di promozione, annotare eventuali riferimenti intragenerazionali in una lista. La ricerca non sarebbe più fatta scorrendo tutti gli oggetti della vecchia generazione ma limitandosi ai soli che abbiano subito modifiche. Questi ultimi potrebbero essere identificati da un flag nella struttura dell'oggetto, oppure si potrebbe gestire l'evento sfruttando la protezione in scrittura dello heap ogni volta che si modifica un oggetto della vecchia generazione. [IBM]

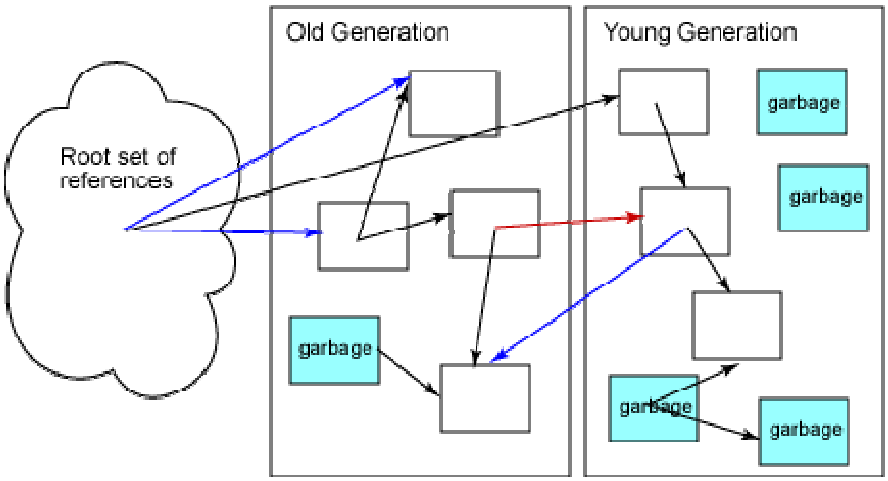

Figura 2. 8 Collector Generazionale

Dati statistici confermano che le modifiche che portano un oggetto della nuova generazione a riferirne uno della vecchia sono molto poche. Quando accadono una possibile azione da intraprendere è quella di far retrocedere l'oggetto riferito dalla nuova alla vecchia generazione.

2.2.3.5 Un esempio

La tecnica di Garbage Collection che la Sun Microsystem ha adottato nella *Java HotSpot Virtual Machine*[*], è di *copia*, *generazionale*, di tipo *preciso* (non conservativo) e incrementale [HotSpot]. Suddivide lo heap nelle due aree fisiche corrispondenti alla nuova e vecchia generazione.

Per rendere più brevi le collezioni minori si evita di percorrere la vecchia generazione in cerca di puntatori alla nuova grazie ad una struttura nota come *tabella delle carte*. La vecchia generazione è suddivisa in parti da 512 byte note come *carte*. Ogni volta che si accede in scrittura a una carta si setta un flag di *dirty*. Durante una collezione minore sono scorse solo le aree con tale flag settato per cercare riferimenti intragenerazionali, diminuendo di gran lunga il peso di tale ricerca.

[*] E' la macchina virtuale di produzione della Sun Microsystems

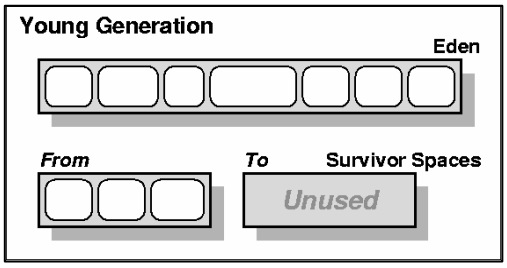

Figura 2. 9 Area della nuova generazione

La nuova generazione è suddivisa in tre parti:

- L'**Eden** contiene la maggior parte dei nuovi oggetti allocati (quelli di dimensioni molto grandi potrebbero essere messe direttamente nella vecchia generazione). L'Eden si svuota sempre dopo una collezione minore.
- I due **Spazi dei Sopravvissuti** (**Survivor Spaces**), che tengono gli oggetti che sono, appunto, sopravvissuti ad almeno una collezione minore, ma a cui è stata data un'altra possibilità di morire prima di essere promossi alla vecchia generazione. Delle due, come si vede in figura 2.8, una sola tiene gli oggetti mentre l'altra è inutilizzata.

Durante una collezione minore gli oggetti individuati "vivi" nell'Eden (quelli che sono stati trovati essere "garbage" sono indicati in figura 2.10 con una 'X') vengono copiati nello spazio inutilizzato. Vengono inoltre copiati quelli dello spazio dei sopravvissuti ai quali si vuole dare un'altra possibilità di morire. Gli oggetti che risultano "abbastanza vecchi" (sono sopravvissuti un numero prestabilito di volte) sono invece promossi nella *Old Generation*.

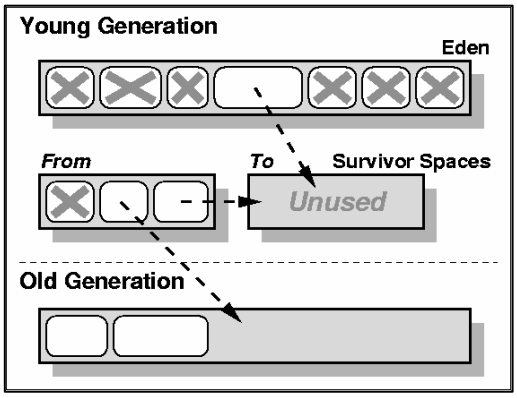

Figura 2. 10 Durante una collezione minore

Alla fine della collezione minore i ruoli dei due Spazi dei sopravvissuti si invertono (gli oggetti ritenuti "garbage" perdono di significato) e l'Eden risulta svuotato.

Nuovamente solo uno dei due spazi è in uso, e l'occupazione della vecchia generazione è leggermente cresciuta. Questo svuotamento risulta molto efficiente perché rende contigua l'allocazione dei nuovi oggetti

.

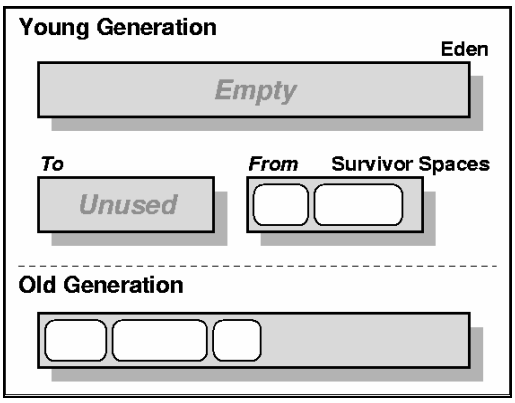

Figura 2. 11 Situazione dopo una collezione minore

3. Applet per Java Card

3.1 La larga diffusione delle Smart Card

L'idea di incorporare un circuito integrato in una carta di plastica fu introdotta, per la prima volta nel 1968 da due inventori tedeschi, Jürgen Dethloff e Helmut Grötrupp che, poco dopo, brevettarono questa loro invenzione. Indipendentemente Kunitaka Arimura dell' Arimura Technology Institute brevettò le smart card nel 1970. Bisognerà comunque aspettare i 47 brevetti siglati in 11 stati diversi da Roland Moreno tra il 1974 e il 1979 per avere un vero progresso. Alla fine degli anni '70, infatti, CII-Honeywell-Bull (adesso il gruppo Bull) [Bull], commercializzarono per primi la tecnologia smart card e introdussero le carte con microprocessore.

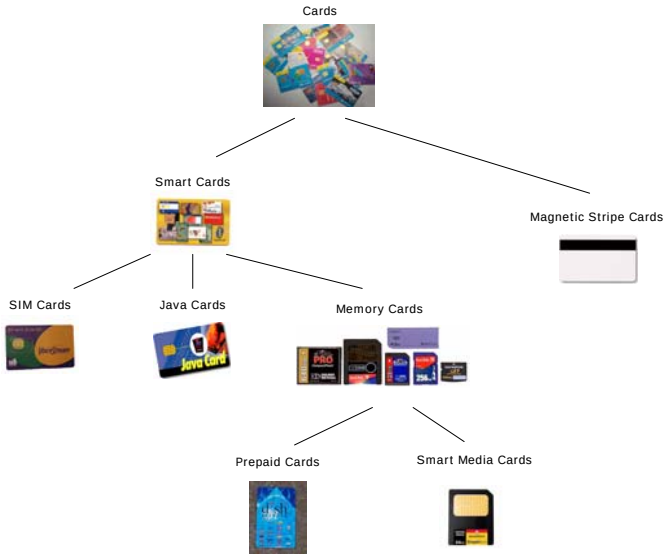

Figura 3. 1 Alcune famiglie di carte attualmente in commercio

I punti di forza delle smart card sono:

- Le ridotte dimensioni e la resistenza agli urti, che la rendono facilmente trasportabile.
- La facilità d'uso.

- La capacità di memorizzare informazioni, che la rende adattabile a tantissime applicazioni.
- La sicurezza, in quanto le informazioni in essa contenute non dipendono da risorse esterne: per rubare le informazioni è necessario il possesso fisico della carta, oltre che una approfondita conoscenza dell'hardware e del software ed attrezzature particolari.

Gli impieghi più comuni per le smart card sono le carte telefoniche prepagate e quelle con servizi GSM per la telefonia mobile, le carte di credito elettronico, atte a rimpiazzare l'uso della cartamoneta, la carta di identità elettronica con dati e informazioni personali, la carta per decodificare segnali satellitari fornita come abbonamento alla Pay-per-View o quelli digitali terrestri che sta avendo luogo proprio in questi ultimi tempi.

Sviluppare un'applicazione per smart card è sempre stato, purtroppo, un processo lungo e difficile. Anche se c'è uno standard nella forma e dimensioni e nel protocollo di comunicazione, gli applicativi sono diversi da un produttore all'altro. Infatti gli strumenti utilizzati per lo sviluppo sono quelli specifici forniti dai costruttori della carta che usano un linguaggio generico ed emulatori hardware dedicati, ottenuti a loro volta dai venditori del chip presente sulla carta. Ne risulta che è virtualmente impossibile che terze parti sviluppino applicazioni in maniera indipendente. Inoltre l'applicazione presente sulla smart card è introdotta in fase di fabbricazione, e non c'è modo di aggiungerne di nuove soprattutto se appartenenti a produttori diversi.

3.2 Architettura delle Smart Card

Le smart card furono una evoluzione delle carte magnetiche, le specifiche prevedono, infatti, che vi sia ancora lo spazio necessario alla striscia magnetica e l'area per il rilievo. Esse in più aggiungono alle capacità ridotte di memorizzazione anche quelle computazionali grazie ad un microprocessore inserito nel substrato di plastica. La comunicazione con l'esterno avviene attraverso otto contatti metallici (*contact card*).

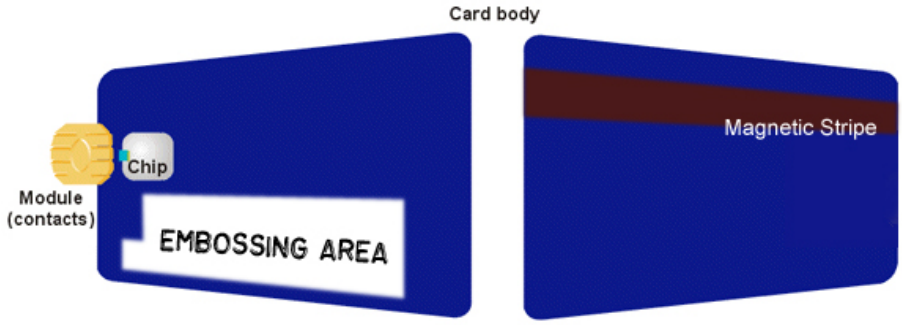

Figura 3. 2 Struttura Smart Card

Esistono poi le *contactless* card, dove non sono presenti i suddetti contatti ma c'è una antenna interna alimentata a batteria. Quest'ultimo tipo di carte è meno diffuso a causa dello scarso raggio di azione, della limitatezza nella quantità di dati inviati, della scarsa sicurezza nella protezione dei dati e, infine ma non per importanza, il loro costo di produzione elevato.

Il processore più largamente utilizzato è un microcontrollore a 8-bit con set di istruzioni del Motorola 6805 o Intel 8051. Il clock varia dai 5 Mhz fino a 40 Mhz per le carte più sofisticate e dotate di moltiplicatore di frequenza. Le Java Cards di ultima generazione hanno microcontrollori a 16 e 32 bit, hanno un moltiplicatore per il clock e cominciano ad essere sviluppate anche su chip con architettura RISC.

La memoria nelle smart card (e anche nelle Java Card, di cui poi ne verrà approfondito l'utilizzo) è elettronicamente di tre tipi:

- **ROM (Read Only Memory)**. E' di sola lettura, viene utilizzata per memorizzare i dati in maniera persistente anche con lo spengimento dell'alimentazione. Le informazioni che contiene sono di solito dati di fabbrica della carta, informazioni definitive dell'utente che ne verrà in possesso, o anche solo alcune routine di sistema. Molto comunemente utilizzata la cella a *MOSFet*, questo tipo di memoria conta una densità di realizzazione elevata e di costi molto ridotti. Un dimensionamento tipico di una smart card è dato da circa 24Kbytes di memoria ROM.
- **EEPROM (Electrically Eraseable Programmable ROM)**. Come la ROM anch'essa mantiene i dati in modo permanente. La differenza è che è possibile anche scrivere all'interno di tali memorie, la cella a doppio *CMOS* con uno di tipo *FLOTOX* (*Floating Gate Thin Oxide Mos*) può essere infatti programmata elettricamente. I tempi di lettura sono paragonabili a quelli delle ROM (100-

300 ns), mentre quelli di scrittura risultano 1000 volte più lenti (100-300 µs). Tali scritture, inoltre, non possono avvenire per più di 100000 volte e il tempo di permanenza dell'informazione di 10 anni (dato comunque rassicurante) è dovuto al deperimento dello strato di ossido dei FLOTOX. Questo tipo di memoria risultano non solo più costose delle controparti non programmabili, ma anche molto ingombranti, rendendone limitata la loro dotazione nei modelli di smart card in commercio (da 16Kbyte al Mbyte nei modelli più sofisticati). Queste caratteristiche rendono però la EEPROM il tipo di memoria giusta per memorizzare le applicazioni poste su carta sia durante che dopo la produzione in fabbrica. L'innovazione in questo tipo di memoria è l'utilizzo di FLASH-EEPROM, che offrono una maggiore densità realizzativa (la cella è a singolo transistor) e tempi di accesso inferiori al prezzo di una programmazione elettrica a blocchi anziché dei singoli bit.

- **RAM (Random Access Memory)**. E' un tipo di memoria temporanea (si perdono i dati in caso di reset) con tempi di lettura e scrittura rapidi (100ns circa) che possono avvenire infinite volte. Hanno una scarsa densità realizzativa (la cella è un bistabile a 6 transistor) ed un discreto costo realizzativi. La dotazione sulle carte è media e si aggira tra 1 e 4 Kbytes. Viene usata per memorizzare i dati temporanei delle applicazioni, come ad esempio il buffer di comunicazione con l'esterno.

3.3 Modalità e Protocollo di Comunicazione

La comunicazione con le smart card avviene mediante l'utilizzo di appositi lettori noti come Card Acceptance Device (CAD). La carta è inserita o avvicinata (nel caso di *contactless card*) al CAD. Il CAD può essere connesso direttamente ad un PC, nel qual caso si parla di *lettori*, oppure può essere un computer egli stesso, e questo è il caso dei *terminali*.

Gli applicativi che comunicano con le carte sono le *host applications* e stanno sempre dal lato del lettore e devono prendere iniziativa per interrogare la carta ed attendere la sua risposta. Il modello di comunicazione è, infatti, di tipo *half-duplex* con modalità *master/slave* dove si scambiano unità informative o **Application Protocol Data Unit** (**APDU**). Le richieste sono effettuabili esclusivamente dal lato

CAD il quale invia **Command-APDU**, a cui seguono le risposte effettuate dalla carta sotto forma di **Response-APDU**.

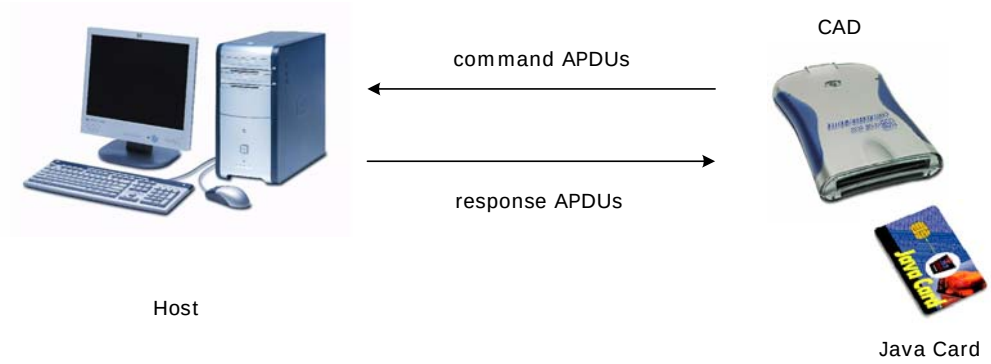

Figura 3. 3 Comunicazione tra host e Java Card

La *C-APDU* presenta una struttura con un *header* obbligatorio di 4 bytes dato dai campi **CLA** (**Class of Instruction**), **INS** (**Instruction Code**) e **P1** e **P2**, col quale si indica il tipo di istruzione richiesta tra quelle supportate passando eventuali parametri.

A tale *header* segue un corpo opzionale che, se contiene dati, comincia con il byte Lc indicante il numero di byte spediti e, se sono richiesti dei dati indietro, termina con il byte Le indicante il numero di byte attesi.

La *R-APDU* è costituita da un corpo opzionale contenente i dati ed un *trailer* obbligatorio formato da due ***Status Word*** (**SW1** e **SW2**) dove si indica l'esito dell'operazione. '90 00', ad esempio, indica un successo mentre un codice che comincia per '6' fa intendere che si sia verificata una eccezione o un errore. In quel caso il codice può dare una informazione circa le sue cause.

Header (obbligatorio)				**Corpo (opzionale)**		
CLA	INS	P1	P2	Lc	Dati (0-255 byte)	Le

Tabella 3. 1 Command APDU

Corpo (opzionale)	**Trailer (obblig.)**	
Dati (0-255 byte)	SW1	SW2

Tabella 3. 2 Response APDU

3.4 L'avvento delle Java Card

I primi a riconoscere le potenzialità che il Java poteva avere nell'ambiente delle smart card furono un gruppo di ingegneri della Schlumberger's [Axalto] che in Austin, Texas nel 1996, proposero una scelta delle API da implementare nelle Java Card. Pochi mesi dopo altre due società, la Bull e la Gemplus [Gemplus], si unirono alla ditta texana per fondare il Java Card Forum [JCForum].

Nel 1997 la Sun Microsystems, in collaborazione con il gruppo del Java Card Forum, annunciò il rilascio delle specifiche Java Card 2.0: questo insieme di APIs è attualmente lo standard per le Java Card. L'ultima versione è la 2.2.1 (ottobre 2003) ed è composta da 3 sezioni che descrivono le specifiche di APIs, Runtime Environment e Virtual Machine [JCspec2.2.1].

Il Java Card Runtime Environment (JCRE) è una sorta di sistema operativo che si interpone tra il basso livello e l'hardware della carta e dei suoi metodi nativi e tra l'alto livello delle applicazioni scritte dai programmatori nel linguaggio Java e note come Java Card Applet.[Chen]

I benefici che si traggono sono molteplici:

- Scrivere programmi per Java Card risulta più semplice grazie al linguaggio ampiamente diffuso.
- Gli aspetti della sicurezza del linguaggio Java ben si prestano all'ambiente delle smart card. Come esempio si pensi alle restrizioni di accesso a campi e a metodi che avvengono nel Java. Ad esse si aggiungano le capacità di un firewall realizzato in hardware per comprendere quanto sicuri possano essere i dati sulla carta.
- Le proprietà di indipendenza dall'hardware del Java sono qui ereditate. I programmatori non devono preoccuparsi di che tipo di protocollo utilizzare nella comunicazione o di quale set di istruzioni il microprocessore si avvalga. Tutto ciò che sta al di sotto del JCRE è aggirato dalle chiamate alle Application Program Interfaces (API), le quali svolgono il lavoro a basso livello.
- E' possiblie memorizzare e gestire indipendentemente applicazioni multiple, sia che siano già presenti in fase di produzione (*Preissuance Applets*), sia che siano state scaricate in un secondo momento ed installate sulla carta (*Postissuance Applets*).

- La compatibilità pressoché totale con gli standard delle smart card, grazie al pieno supporto della normativa **International Standard n°7816** (**ISO 7816**).

3.5 Sottoinsieme del linguaggio Java

A causa delle ridotte capacità di memoria e computazionali, la piattaforma Java Card supporta solo un sottoinsieme attentamente scelto di funzionalità del linguaggio Java che tenendo aspetti che più si addicono agli "ambienti ristretti" delle smart card pur preservando le caratteristiche del linguaggio orientato agli oggetti.

La tabella 3.3 riassume quali caratteristiche siano preservate e quali invece sono non supportate.

Caratteristiche supportate	Caratteristiche non supportate
• tipi primitivi "piccoli": boolean, byte, short • array uni-dimensionali • packages, classi e interfacce Java • caratteristiche object-oriented di Java: eredità, metodi virtuali, overloading, creazione dinamica degli oggetti, ... • il supporto per il tipo int e per gli interi a 32 bit è opzionale • eccezioni	• i tipi primitivi long, double e float • caratteri e stringhe • array multi-dimensionali • caricamento dinamico delle classi • clonazione degli oggetti • threads • security manager • garbage collector • finalizzazione degli oggetti

Tabella 3. 3 Funzionalità Java supportate o meno

3.6 La Piattaforma Java Card

La piattaforma Java Card è divisa in tre parti:

- La **Java Card Virtual Machine (JCVM)**
- Il **Java Card Runtime Environment (JCRE)**
- Le **Java Card Application Programming Interfaces (API)**

3.6.1 La Java Card Virtual Machine (JCVM)

Una prima differenza tra la Java Card Virtual Machine (JCVM) e la Java Virtual Machine (JVM), è che la JCVM è implementata in due parti separate: una *off-card* ed una *on-card*.

La parte *off-card* è il *convertitore* (*converter*) e si occupa di trasformare il *bytecode* dei file class in un nuovo formato, chiamato CAP (*converted applet*): il CAP file verrà poi caricato sulla Java Card ed eseguito dall'interprete. La conversione è necessaria per ottimizzare il codice del file class (nei sistemi embedded la memoria è preziosa perché poca) e per compiere le verifiche statiche e strutturali sul codice (verifica off-card).

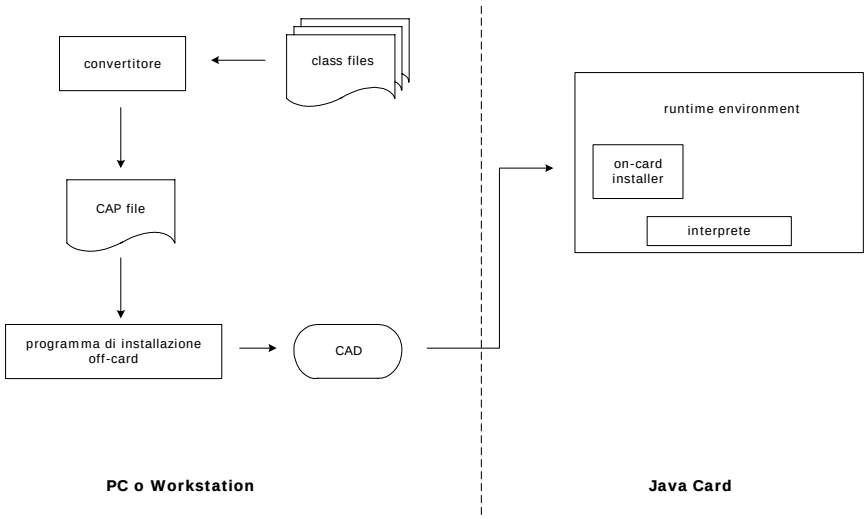

Figura 3. 4 Conversione dei File Class e Java Card Installer

La parte *on-card* della Virtual Machine fornisce il supporto al *run-time* caratteristico del modello Java, offrendo quindi una macchina virtuale che consente l'indipendenza tra il codice delle *applets* e l'hardware utilizzato.

3.6.2 Il Java Card Runtime Environment (JCRE)

Il run-time environment delle Java Cards (JCRE) è costituito da tutte le componenti del sistema che sono sulla carta:

• le APIs (*framework classes*): sono le funzionalità standard definite nelle specifiche Java Card [JCspec2.2.1]; tutte le *applets* accedono alle funzionalità della carta attraverso queste APIs.

• l'*installer*: serve ad installare il software sulle Java Cards; senza l'*installer* tutto il software delle carte, incluse le *applets*, dovrebbe venire scritto nella memoria della Java Card durante il processo di produzione.

• le estensioni specifiche delle industrie (*industry-specific extensions*): sono delle librerie che possono offrire servizi aggiuntivi per completare ed estendere il sistema ed il modello di sicurezza della Java Card.

• le classi di sistema (*system classes*): costituiscono il *core* del JCRE; offrono tutti gli strumenti necessari per gestire la comunicazione, la creazione ed il controllo delle *applets*.

• i metodi nativi (*native methods*): sono l'insieme di metodi responsabili dei protocolli di comunicazione a basso livello, della gestione della memoria e della crittografia, offrono cioè alla JCVM tutti i metodi necessari per pilotare correttamente l'hardware.

• la JCVM, Java Card Virtual Machine (*bytecode interpreter*). I compiti principali della parte *on-card* della *Virtual Machine* sono:

- esecuzione delle istruzioni del *bytecode*;
- creazione e allocazione degli oggetti in memoria;
- verifica della sicurezza al *run-time*.

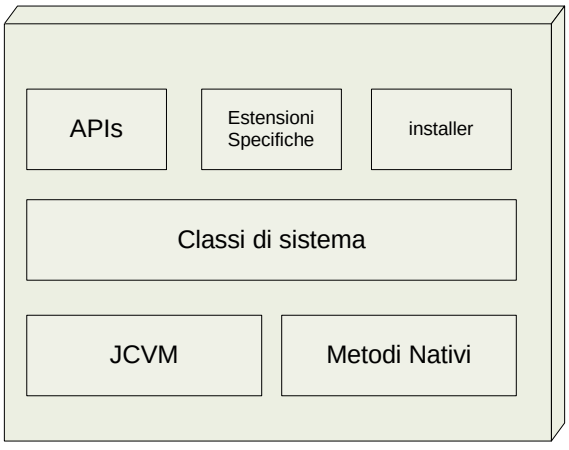

Figura 3. 5 Il Java Card Runtime Environment

Il JCRE è inizializzato una sola volta in tutta la sua vita: all'atto della creazione della carta è inizializzata anche la JCVM e sono creati gli oggetti e le strutture dati atte a fornire supporto e gestione delle applets.

In coincidenza del reset della carta, ad esempio perché si interrompe la sua alimentazione estraendola dal CAD, il JCRE è sospeso e tutti i dati creati sono mantenuti (come sarà meglio illustrato tutti i dati sono nella EEPROM) anche attraverso più sessioni di CAD. Alla riattivazione il JCRE riparte ma non dal punto in cui si era interrotto bensì nel loop principale.

Una sessione di CAD è infatti una serie di operazioni ripetitive che partono dalla fase di alimentazione (*powerup*) a quella di interazione tra la carta e l'esterno attraverso APDU scambiate e, nella carta, tra il JCRE e le applet.

Tre sono le nuove funzionalità offerte dal JCRE:

- *Atomicità e Transazioni*. E' assicurato che ogni operazione di scrittura in un singolo campo di un oggetto o in una classe è atomico. Grazie alle *transaction APIs*, è garantita la coerenza degli oggetti con meccanismi di *commit* e di *rollback* che assicurano che "o la modifica è stata fatta in maniera completa, o non è stata fatta per niente".
- *Oggetti persistenti e transitori*. Esiste una differente gestione degli oggetti temporanei e di quelli a lunga esistenza che viene effettutata grazie a chiamate esplicite fatte dal programmatore (cfr paragrafo omonimo).
- *Firewall delle Applet e meccanismo di condivisione*. Le applet risultano avere il proprio spazio di memoria isolato le une dalle altre, a meno che non vogliano espressamente condividere dei metodi (cfr paragrafo omonimo).

3.6.3 Le Application Programming Interfaces (API)

Le API offerte dalla Java Card consistono in un set di classi atte a programmare le smart card in accordo alle specifiche ISO 7816. Contengono principalmente tre package:

- *Java.lang* è solo un sottoinsieme di quello del Java e supporta classi come *Object* o *Throwable* e qualche tipo di eccezione.

- *Javacard.framework* è essenziale e definisce tutte le classi che servono al funzionamento delle applet sulla carta, la loro creazione con la classe *Applet* e la loro interazione con l'esterno con la classe *APDU*, solo per fare un esempio.
- *Javacard.security* fornisce funzioni crittografiche, come la classe *keyBuilder*, utili a realizzazioni di chiavi sia per algoritmi simmetrici (DES) che asimmetrici (RSA), e utilità pseudo-random per computare firme e *digest*.

3.7 Le Java Card Applet

Da non confondersi con le Java Applet, le Java Card Applet sono dei programmi che aderiscono ad un set di convenzioni che le permettono di girare con il JCRE. Il motivo della loro denominazione (d'ora in poi saranno sempre chiamate semplicemente Applet), è dovuto al fatto che possono essere caricate in momenti successivi alla fabbricazione della carta.

Per essere tale un'applet deve derivare direttamente dalla classe *Applet*, appartenente al package *framework*, che è la superclasse di tutte quelle presenti sulla carta e determina l'overloading di alcuni metodi necessari al suo funzionamento.

Ogni Applet è identificata univocamente da una serie di byte nota come **Application Identifier** (**AID**), a sua volta composta da un identificativo aziendale *Resource Identifier* o RID da 5byte e assegnato unicamente dallo standard ISO, e da un identificatore specifico di applicazione *Proprietary Identifier Extension* o PIX (0-11 byte) scelto in maniera opportuna dal proprietario per il particolare package.

Con l'AID un'applet, che si installa sulla Java Card, si registra anche con il Runtime Environment, e sempre con l'AID si seleziona un'applet per farla comunicare con l'esterno.

Scendendo più nel dettaglio [JCspec2.2.1], di seguito sono riportate le firme dei metodi più significativi importate dalla superclasse:

```
import javacard.framework.Applet;
public static void install (byte[] bArray, short bOffset, byte
bLenght);
public boolean select();
public void deselect();
protected void register();
abstract void process(javacard.framework.APDU apdu);
```

- **Install()**, chiamato dalla JCRE, è il metodo col quale l'applet si registra col JCRE, l'argomento passato è il buffer dove sta l'AID, il suo offset e la sua lunghezza. All'interno vi si chiama il costruttore dell'applet per istanziare le proprie strutture dati.
- **Register()** segna l'istante in cui la transazione di installazione è conclusa. E' sempre chiamata all'interno del costruttore.
- **Select()** è chiamato dalla JCRE ogni volta che si riceve una APDU di tipo *select* ed è identificata l'applet a cui è destinata grazie all'AID passata nel campo dati della C-APDU.
- **Deselect()** è chiamata automaticamente dalla JCRE quando selezionando una nuova applet, si deseleziona quella che prima era attiva
- **Process()** è il metodo che deve gestire la comunicazione con l'esterno attraverso le APDU

Il tempo di vita di un'applet sulla Java Card comincia quando essa è creata per mezzo del metodo `Install()`. Si apre una transazione, viene creata una istanza dell'applet, il costruttore alloca le strutture dati, e finisce con il metodo Register (*commit* della transazione). In quel momento il JCRE è a conoscenza dell'esistenza dell'applet la quale si trova in uno stato inattivo. Quando arriva una APDU il JCRE controlla se non si tratti di una selezione. In tal caso scorre l'AID richiesto su una tabella che possiede e chiama il metodo `select()` dell'applet corrispondente, la quale passa in uno stato attivo*. Il metodo `process()` gestisce l'APDU trasmessa e tutte le successive. Quando arriva una APDU con una richiesta di selezione di una nuova applet, viene invocato il metodo `deselect()` e l'applet da attiva torna nello stato inattivo. Anche l'estrazione della carta dal CAD porta l'applet in esecuzione da

* In ogni momento esiste sempre e solo una applet attiva alla volta

attiva allo stato inattivo. Questo meccanismo è illustrato nel diagramma di stato di figura.

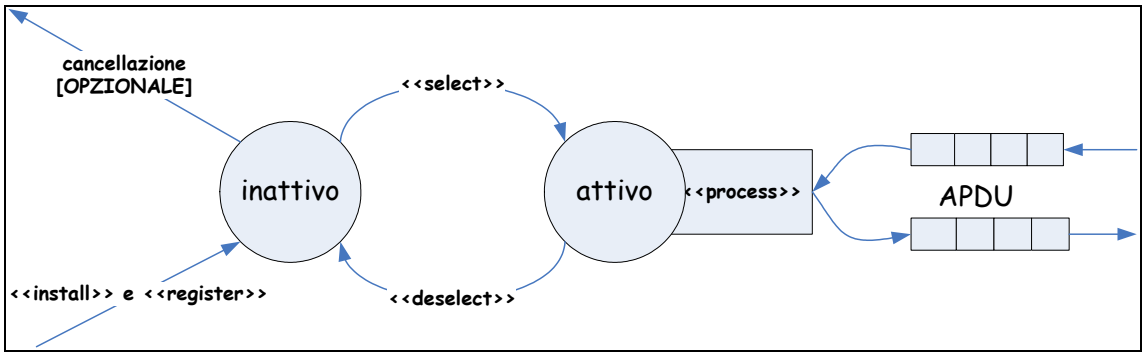

Figura 3. 6 Diagramma di Stato di un'applet

3.8 Modello di memoria delle Java Card

In questa sezione sono riportati gli strumenti messi a disposizione dal JCRE per operare sulla memoria.

Un tipico sistema Java Card prevede che il codice del runtime environment, contenente la virtual machine, le classi API ed altro software, sia tutto inserito all'interno della ROM. Tale tipo di memoria non più modificabile ben si presta a questo utilizzo. Già nelle smart card vi venivano memorizzate informazioni come il PIN, dati di fabbrica, dati personali. E' possibile mettere nella ROM anche codice di applet che, magari, appartiene ad uno standard e non ha necessità di modifica.

La RAM invece è una memoria temporanea ideale per contenere lo *stack a runtime*, così come risultati intermedi di computazioni dei metodi nativi di crittografia, parametri dei metodi e variabili locali.

Tutto il resto come le applet scaricate, gli oggetti che esse usano e creano attraverso l'operatore "new", va a risiedere nella EEPROM.

E' una scelta di progetto molto restrittiva ma necessaria, in quanto sarebbe impensabile che con una improvvisa cessazione dell'alimentazione alcuni degli oggetti dell'applet (magari anche membri della classe) venissero perse ed altre invece no. Ci sono comunque degli oggetti che il programmatore sa a priori che non serve siano tenuti per sempre in memoria. Ad esempio variabili di un ciclo di "for", o un buffer di

comunicazione. E' per questo che la versione 2.1 e successive delle specifiche delle Java Card prevedono l'esistenza di due tipi di oggetti: quelli **persistenti (Persistent Objects)** e quelli **transitori (Transient Objects).**

Gli oggetti *persistenti* hanno le seguenti proprietà:

- Sono creati dall'operatore *new.*
- Mantengono i valori tra sessioni diverse di CAD.
- Ogni aggiornamento fatto su un oggetto persistente è di tipo *atomico*, cioè non interrompibile, e un eventuale improvvisa cessazione nel mezzo porta all'annullamento di tutto l'aggiornamento ed è ripristinato il vecchio valore.
- Un oggetto persistente può essere riferito da un campo di un oggetto transitorio.
- Un campo di un oggetto persistente può riferire un oggetto transitorio.
- Se un oggetto persistente non è riferito da nessun altro oggetto, diviene irraggiungibile e può essere *garbage collected*.
- Tutti gli oggetti persistenti risiedono nella EEPROM.

Gli oggetti *transitori* non sono da confondersi con i campi definiti nel Java dalla parola chiave "`transient`", i quali servono solo a indicare cosa non deve far parte della serializzazione di un oggetto[*]. Gli oggetti trasitori sono caratterizzati dalle seguenti proprietà:

- Sono possibili due tipi di oggetti transitori:
    - Un array di primitivi (boolean, byte, short)
    - Un array di riferimenti a oggetti (Object)
- Vengono creati attraverso chiamate a JC API che creano l'array ritornando il puntatore all'oggetto.
- Non contengono dati durevoli tra più sessioni di CAD.
- Ogni aggiornamento che viene fatto su un oggetto transitorio non è atomico
- Possono essere riferiti da un campo in un oggetto persistente.
- Se non è più raggiungibile verrà *garbage collected.*
- Le scritture su oggetti transitori non avranno problemi di performance in quanto tali oggetti risiedono nella RAM della Java Card.

Esistono due tipi di oggetti transitori. Il tipo di un oggetto va indicato al momento della invocazione della corrispondente API:

[*] Tale parola chiave non è, infatti, neppure presente nel sottoinsieme del linguaggio Java adottato dalle Java Card.

1. **CLEAR_ON_RESET**: mantengono il proprio valore tra due o più selezioni di applet
2. **CLEAR_ON_DESELECT**: mantengono il proprio valore solo all'interno di una sola selezione.*

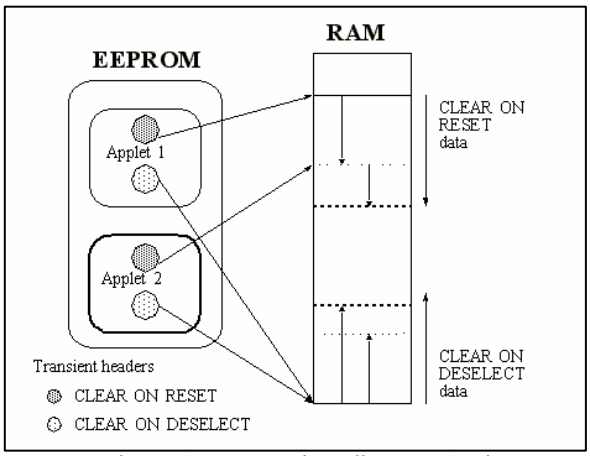

Figura 3. 7 Memoria nelle Java Card

Dalle specifiche della Java Card [USENIX], la RAM risulta suddivisa in due parti, quella riservata agli oggetti CLEAR ON RESET e quella per i CLEAR ON DESELECT che, ognuno ad un estremo, crescono al centro (il JCRE deve cercare di evitare sovrapposizioni).

L'informazione circa la locazione e la dimensione di ognuno dei due tipi di memoria è contenuta in un *header* persistente all'interno di ogni applet che la utilizzi: ovviamente l'area CLEAR ON DESELECT di un'applet, "ripulita" ad ogni nuova selezione, si va a sovrapporre a quella delle altre applet, mentre l'area CLEAR ON RESET è tenuta separata (si veda la figura 3.5).

Questa particolare funzionalità dell'ambiente Java Card sembra snaturare lo stile di programmazione Java che tiene completamente all'oscuro il programmatore di dove vengano allocati gli oggetti da lui creati. Java, è un ambiente di tipo persistente, ma il Java Card deve necessariamente far fronte a questa opportunità di creare oggetti transitori: oggetti con tempi di vita limitati nella RAM fanno risparmiare spazio nella EEPROM, inoltre necessità di continui accessi a tali oggetti sono più veloci che nella EEEPROM e non ne sprecano gli utilizzi contati (cfr paragrafo 3.2).

* NOTA: un reset causa anche la deselezione dell'applet corrente

3.9 Firewall e meccanismo di condivisione

La piattaforma Java Card è un ambiente **multi-applicazione**: più applet possono coesistere sulla carta e ne possono essere scaricate ulteriori in momenti successivi. Una applet spesso detiene informazioni importanti come un credito bancario, una chiave crittografica privata, per cui si capisce bene come il meccanismo di condivisione dei dati debba essere attentamente limitato. Nella Java Card ad ogni applet risulta assegnato un contesto che è reso isolato grazie al meccanismo di *applet firewall*. Questo contesto è in realtà un contesto di gruppo assegnato ad ogni package. Anche al JCRE è assegnato un proprio contesto ma risulta avere maggiori privilegi.

Il JCRE tiene traccia sia dell'applet correntemente selezionata, sia di quella correntemente **attiva**. Il contesto di tale applet è riferito da una variabile di sistema nota come **contesto attivo** (*active context*).

Ad ogni momento, dunque, c'è un solo contesto attivo; quando è creato un nuovo oggetto la JCRE assegna la sua proprietà (*ownership*) al contesto che in quel momento risulta attivo: se in quel momento è in esecuzione il JCRE allora l'oggetto sarà un oggetto di sistema.

Array di tipi primitivi e statici (dichiarati con la parola chiave `static`) sono di proprietà del contesto di gruppo del package in cui sono dichiarati (sono infatti dichiarati e inizializzati dal *converter*).

Ad ogni accesso di un oggetto si attiva un controllo dei diritti, fatto dalla virtual machine, che va a verificare se il contesto proprietario dell'oggetto sia lo stesso di quello correntemente attivo. Se così non fosse e se il contesto attivo non è il JCRE (che può accedere ad oggetti di contesti diversi) allora l'accesso è negato. Esistono comunque dei meccanismi di condivisione di oggetti e metodi che possono provocare un **cambio di contesto** quando tali oggetti sono acceduti o tali metodi risultano invocati. Il contesto attualmente attivo è salvato e quello nuovo diventa il nuovo contesto attivo: il metodo invocato è ora eseguito nel nuovo contesto e possiede tutti i diritti di quello corrente. Al suo ritorno sarà ripristinato il vecchio contesto e i diritti torneranno quelli originariamente posseduti. Un esempio del meccanismo è dato dall'arrivo di una APDU: gestita dal JCRE si ha l'invocazione dei metodi `process()`, `select()` e `deselect()` dell'applet selezionata.

La piattaforma Java Card mette a disposizione tre tipi di meccanismi di condivisione:

- Gli **oggetti *entry point***, che, appartenenti alla JCRE, risultano accessibli da tutte le applet e vengono usati per fare chiamate di sistema. Essi scatenano un cambio di contesto e passano il comando al JCRE. Ne esistono di due tipi che si differenziano dall'utilizzo che gli utenti ne possono fare nelle proprie applet:
  - *Temporanei* non sono memorizzabili in oggetti locali, in variabili istanza o in array (neppure gli array transitori). Un esempio è la variabile "*javacard.framework.APDU*".
  - *Permanenti* che invece si possono liberamente copiare, come la variabile "*javacard.framework.AID*".
- Gli **array globali**, designati dalla JCRE ed accessibili da tutte le applet in maniera diretta.
- Il **meccanismo di Interfaccia Condivisibile (*Shareable Interface*)**, mediante il quale una applet può, mediante una interfaccia, rendere disponibili alle altre applet dei metodi invocabili e delle variabili non modificabili.

Un oggetto di una classe che implementa una interfaccia condivisibile deve derivare necessariamente dalla classe "*javacard.framework.Shareable*" e prende il nome di *Shareable Interface Object* (*SIO*). Per accedervi le applets devono servirsi solo dei metodi di interfaccia, e il modello che regola tale accesso è di tipo client-server come illustrato dai seguenti passi facenti riferimento alla figura 3.8:

1. L'applet Client deve ottenere anzitutto l'AID dell'applet Server, questo lo si fa grazie al metodo

   ```
   public static AID JCSystem.lookupAID();
   ```

2. E' scatenato un cambio di contesto grazie al quale la JCRE, guardando all'interno di una tabella, ritorna l'AID del sever.
3. Il Client può allora richiedere lo SIO del server invocando la

   ```
   public static Shareable
   jcSystem.getAppletShareableInterfaceObject()
   ```

4. Il contesto è di nuovo cambiato, e passa alla JCRE che, in quanto privilegiata, accede al metodo del Server noto come

   ```
   public Shareable jcSystem.getShareableInterfaceObject()
   ```

5. Un nuovo cambio di contesto permette all'applet Server di effettuare dei controlli su dei parametri passati e, nel caso, di ritornare un riferimento all'oggetto SIO.
6. L'oggetto SIO se passato è preso dalla JCRE e ritornato al Client
7. D'ora in poi il Client potrà invocare i metodi contenuti in tale oggetto.

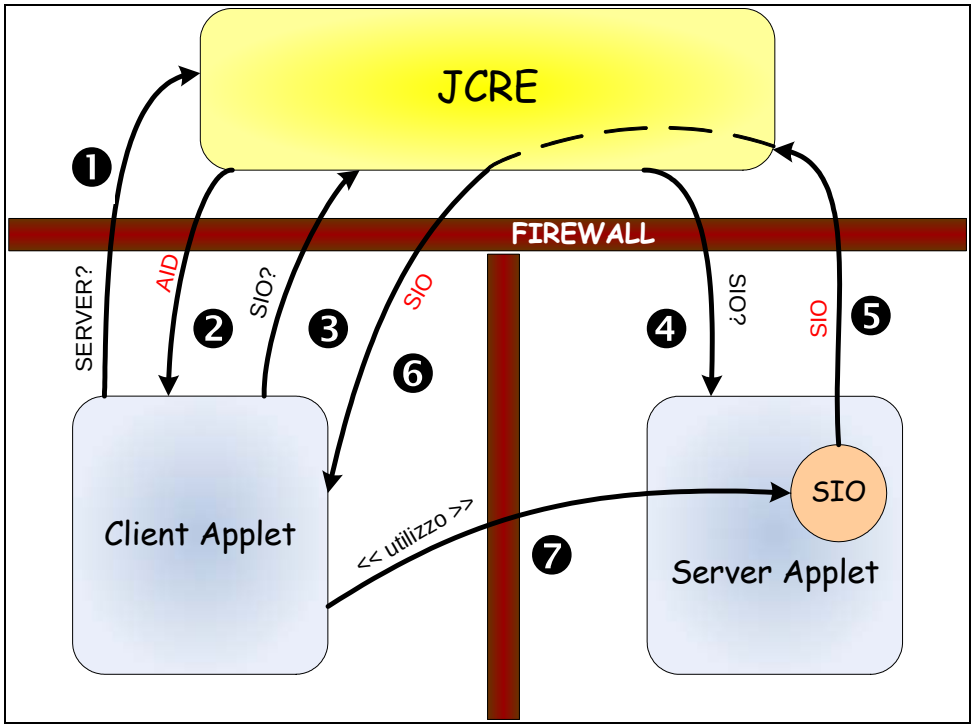

Figura 3. 8 Passi per la condivisione di un SIO

4. Progetto realizzativo

Fino a questo momento è stata fatta una panoramica delle Java Card, del sottoinsieme di funzionalità che eredita dal Java e di quelle che di nuove aggiunge. Sono state inoltre mostrate le caratteristiche fisiche del supporto derivanti dal retaggio di smart card, che hanno sollevato il problema della scarsezza di risorse, e gli strumenti messi a disposizione per meglio gestirle. Sono stati inoltre presentati i principali tipi di Garbage Collectors, da quelli storici fino a quelli oggi implementati nelle ultime release del Java Developer Kit. Per ogni algoritmo è stato evidenziato il funzionamento e i principi su cui si basa, oltre che i pregi ed i difetti.

Tutto ciò è servito ad introdurre l'ambiente in cui si colloca il lavoro svolto in questa tesi: la realizzazione di un garbage collector per Java Card.

4.1 L'ambito di collocamento

Sebbene le specifiche della Java Virtual Machine non prevedono l'utilizzo di un algoritmo di collection in particolare da usare, quelle delle Java Card non prevedono neppure l'obbligatorietà della presenza di tale strumento. Ciò che è certo è che il *Garbage Collector* si debba collocare all'interno della Virtual Machine nel Runtime Environment, dove è possibile accedere alle strutture dati interne che sono ben note ai progettisti ed è possibile implementare il particolare algoritmo proprio a partire da tali strutture.

Per fare un esempio il Java Hot Spot* [Hot Spot], tra una scelta di ben sei in totale, utilizza di default un Garbage Collector di tipo incrementale, non conservativo e generazionale. Le modifiche ai puntatori contenuti nei campi delle vecchie generazioni sono individuate attraverso una variante ottimizzata di un algoritmo nota come *card marking* (segna carte). Lo heap risulta diviso in un set di *carte*, ognuna delle quali inferiore alla pagina in memoria. Nella JVM è mantenuta una mappa delle carte con un

* componente chiave del *core* della Java 2 Platform, Standard Edition (J2SE) della Sun Microsistems.

bit di dirty per ognuna stante a indicare se tale area di memoria è stata acceduta di recente oppure no (cfr Paragrafo 2.2.3.5).

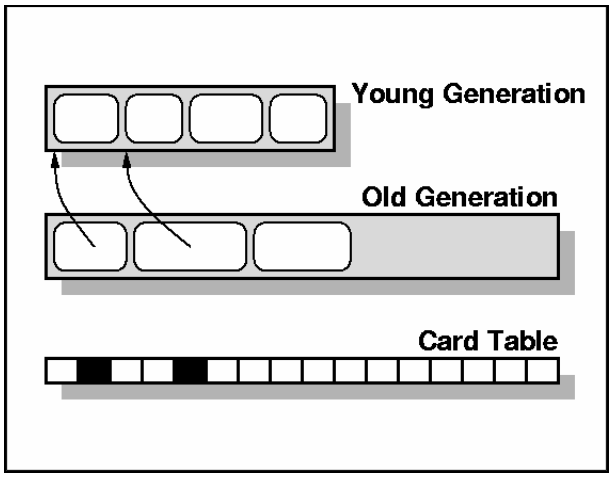

Figura 4. 1 La tabella delle Carte per la vecchia generazione

Nelle Java Card la situazione è simile: solo a livello di JCRE è possibile manipolare il codice in modo da implementare una Garbage Collection di qualsiasi tipo. Poiché il Runtime Environment è collocato sulla carta al momento della fabbricazione, è risultato impossibile agire fino a tale livello con gli strumenti messi a disposizione per il prototipo realizzato in questa tesi.

E' stato creato un prototipo funzionante realizzandolo come una Applet la quale, installata sulla Java Card, agisca "simulando" le azioni di Garbage Collection intraprese dalla Virtual Machine. Le collezioni non sono effettuate sulla memoria ma su un modello che di essa si è costruito quando le altre applet sono in esecuzione. Ciò non sminuisce, quindi, l'importanza didattica del lavoro di tesi.

4.2 Le idee alla base

Porsi a livello di applet, purtroppo, introduce parecchie limitazioni. Una fra tutte è il fatto di non sapere cosa le altre applet stiano facendo sui loro oggetti a causa del *firewall* di sistema che separa le aree di memoria assegnandone una ad ogni package (vedi paragrafo 3.9). Il progetto di creare l'applet Garbage Collector sembrerebbe impossibile già in partenza!

Il meccanismo di *Object Sharing* e il concetto di *write-barrier* vengono incontro a questa mancanza e rendono possibile la realizzazione dell'applet Garbage Collector, d'ora in poi "**gcApplet**".

Una write-barrier è un segmento di codice che viene eseguito automaticamente dal sistema quando accade una scrittura su un particolare oggetto. Tale oggetto si dice essere "dietro una write barrier, (una barriera in scrittura)" o, più informalmente "write barrier-ed (barricato)". Esiste anche una controparte nota come read-barrier, dalle ovvie caratteristiche, che qui non verrà presa in considerazione [GCFAQ].

Per avere un'idea del ruolo che le write-barrier hanno in un garbage collector basta rifarsi all'esempio dell'algoritmo di *card marking* poco prima citato. Per tenere aggiornata la bitmap dei bit di dirty basta porre la *carta* dietro una write-barrier: ogni accesso in scrittura a tale area di memoria sarà intercettato dall'interprete del bytecode, coadiuvato dal compilatore *just-in-time* (JIT), e provocherà anche il settaggio del bit di dirty [devX]. La tabella delle carte, risulta quindi aggiornata automaticamente ad ogni accesso in scrittura.

Il funzionamento del prototipo è garantito dal fatto di porre dietro una write-barrier tutta la memoria del contesto acceduta da una applet. Si impone quindi, che ogni volta che si accede in scrittura alla memoria per la creazione o modifica di un oggetto, ne sia data anche una notifica alla **gcApplet**. Di norma le write-barrier si impongono a livello di sistema, e tutto ciò dovrebbe avvenire in maniera automatica. La gcApplet, invece, gira a "livello utente" (è una applet): dovranno quindi essere le altre applet a notificare gli accessi nel loro codice dopo l'istruzione che modifica la memoria[*]. Si ha così una sorta di write-barrier esplicita.

Per interagire con la gcApplet il punto di accesso dovrà essere uno Shared Interface Object (cfr Paragrafo 3.9) messo opportunamente a disposizione dalla gcApplet.

Come verrà chiarito nel prossimo capitolo si tratta di far inserire ai programmatori delle chiamate ai metodi messi in condivisione dalla gcApplet ogni volta che essi vanno ad operare sulla memoria creando e modificando (assegnando) oggetti.

[*] La notifica può accadere anche prima dell'istruzione che modifica la memoria.

4.3 L'algoritmo implementato

Tra tutti gli algoritmi di Garbage Collection quello che è stato implementato nel prototipo è una variante di tipo non conservativo e incrementale del *Reference Counting*.

La gcApplet è di tipo preciso perché il modello che simula la memoria contiene una ulteriore struttura ad hoc sa se un oggetto in memoria è un riferimento o meno (cfr Paragrafo 2.2.1). Tale struttura informa anche di quali siano le dipendenze strutturali tra gli oggetti e del loro tempo di vita.

E' incrementale perché se si riesce a tollerare l'*overhead* introdotto da un aggiornamento del modello per ogni nuova creazione o assegnamento, allora la collezione degli oggetti ritenuti "garbage" è un processo che può tranquillamente avvenire a piccoli passi.

L'algoritmo è il *Reference Count* leggermente modificato: ogni oggetto del modello di memoria contiene un contatore che viene incrementato ogni volta che è riferito da un oggetto in più, e viene invece decrementato ad ogni uscita di *scope* o nuovo assegnamento del riferimento. La leggera modifica consiste, come sarà meglio illustrato nel Paragrafo 4.3.1.2, nella maniera di accedere agli oggetti attraverso una particolare struttura dati.

4.3.1 Strutture dati necessarie

Le specifiche non impongono alcun vincolo sulla maniera di rappresentare gli oggetti in memoria né sulle strategie di allocazione. Sta agli implementatori il compito di associare, ai byte che i vari oggetti occupano, tutta una serie di informazioni per permettere l'allocazione, la garbage collection e la deallocazione, l'acquisizione in mutua esclusione, e altre funzionalità che qui non prenderemo in considerazione. Ciò che ne risulta è una scelta implementativa di come strutturare i dati e di come gestirli.

Prima di introdurre le scelte fatte nel prototipo, viene proposta una soluzione implementata attualmente.

4.3.1.1 Esempio di Soluzione attuale

Per fare un esempio delle soluzioni proposte attualmente, la release del Java SDK, dalla versione 1.2.2 sino alla 1.3.1 [IBMGC], si basa su una struttura dati che rappresenta l'oggetto in memoria come evidenziato in figura 4.2:

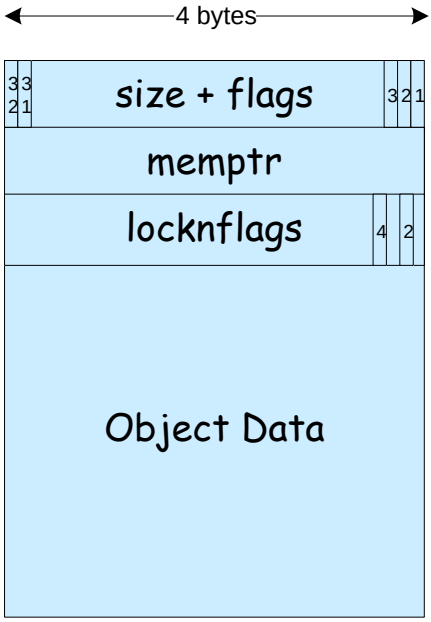

Figura 4. 2 Un Oggetto in memoria

La figura si riferisce ad una macchina con architettura a 32 bit (per quella a 64 bit semplicemente ogni riga occuperebbe 8 bytes). Il significato dei vari campi è di seguito riportato*:

- **Size + flags**. Lo scopo principale è di contenere la lunghezza dell'oggetto. Poiché tutti gli oggetti sono di una lunghezza multipla di 8 byte, gli ultimi tre bit sono liberi e utilizzati per identificare alcuni stati. Inoltre, poiché la dimensione degli oggetti è limitata, si usano anche i 2 bit in cima:
  - **Bit 1**: è il bit di *swap* usato durante la compattazione, il *NotYetScanned* durante la fase di marking, e anche il bit di *multipin* per i pin multipli (vedi bit 3).

* Non vengono fornite spiegazioni più dettagliate perché esula dal contesto della tesi.

  - **Bit 2**: è il *dosed* bit, settato se l'oggetto risulta raggiungibile dallo stack o dai registri (e se quindi appare nel root-set dell'algoritmo di tracing).
  - **Bit 3**: è il bit di *pin*, settato se l'oggetto risulta inamovibile poiché riferito da fuori lo heap.
  - **Bit 31**: è il bit di *FLC* (*Flat Locked Contention*) usato nelle lock per la mutua esclusione.
  - **Bit 32**: è per il valore Hash dell'oggetto, obbligatorio poiché risulta collegato il suo indirizzo in memoria.
- **memptr**. Con granularità di 8 byte, questo slot ha due significati:
  - Se l'oggetto non è un array allora punta a un *blocco dei metodi*, dal quale si può capire di quale classe sia istanza.
  - Se l'oggetto è un array, contiene il numero di entrate nell'oggetto.
- **locknflags**. Usato dal componente *LK* per le lock, contiene anche dei flags col seguente significato:
  - **Bit 2**: il flag di *array* che dice se l'oggetto è un array o meno.
  - **Bit 4**: è il bit di *hased and moved*. E' settato se l'oggetto è stato rilocato dopo un hash. In quel caso il nuovo valore di hash è nell'ultimo slot dell'Object Data.
- **Object Data**. E' dove comincia l'oggetto vero e proprio, la sua struttura dipende dal tipo di oggetto.

La allocazione e la deallocazione degli oggetti va a modificare, inoltre, due ulteriori array di bit che si riferiscono allo heap. Poiché gli oggetti sono di almeno 8 byte allora ogni bit dell'array rappresenta otto bytes dello heap, e lo spazio occupato da ognuno dei due vettori è 1/64 dello heap.

- **allocbits** è un vettore di bit che indica quale aree di memoria risultano occupate dall'allocazione di oggetti. In pratica da questa informazione capiamo dove comincia l'oggetto ma non se questo risulta "vivo"
- **markbits** serve alla fase di marking del Garbage Collector per sapere quali oggetti ha percorso e quali no, e quindi saranno riconoscibili gli oggetti non più referenziati.

La figura rappresenta il caso in cui due oggetti sono stati allocati ma solo il numero 1 risulta ancora "vivo".

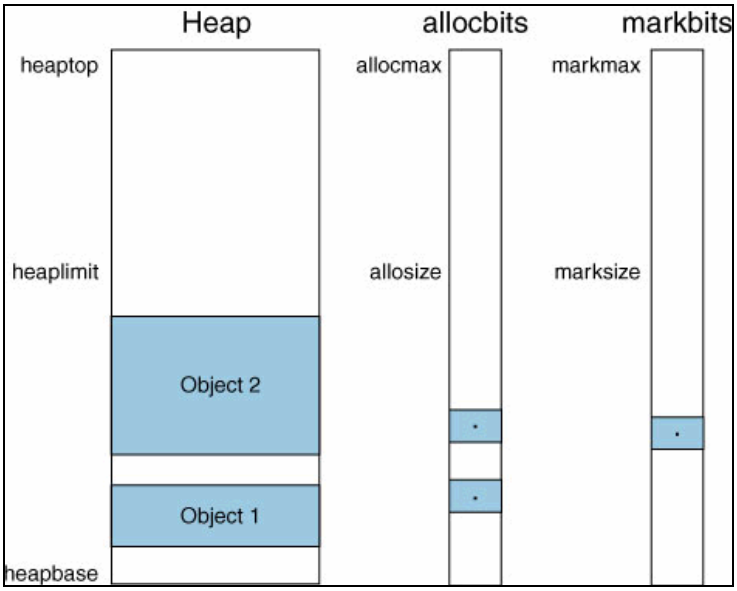

Figura 4. 3 Oggetti nello heap

L'*overhead* totale di memoria occupata è dunque di 1/32 della dimensione dello heap, a cui bisogna sommare i 12 bytes per ogni oggetto. Questa soluzione è inapplicabile nel contesto delle Java Card*, ma è stata riportata per evidenziare le caratteristiche delle strutture dati che ospitano i bytes allocati dall'utente per ogni oggetto. Nel prossimo paragrafo sarà dato modo di confrontare l'occupazione delle strutture dati utilizzate dal prototipo **gcApplet**.

A nota informativa si sappia che dalla versione 1.3.0 e successive è stato implementato lo *heap di sistema*, una zona di memoria non contigua in cui sono allocati gli oggetti il cui tempo di vita è lo stesso della JVM e che quindi non devono essere collezionati.

Le zone di memoria libera sono mantenute da una lista con un campo informativo contenente la dimensione ed un puntatore alla zona successiva. Ci sono molti tipi di allocazione, la più semplice prende in lock tutto lo heap e scorre la lista delle zone libere in cerca di una che la soddisfi.

* Infatti questa soluzione è quella utilizzata nella Java Virtual Machine, dove non si hanno requisiti stringenti di occupazione memoria come invece esistono nel Java Card.

4.3.1.2 Soluzione Proposta

Le scelte progettuali del prototipo hanno dovuto far fronte al fatto di avere risorse molto limitate. L'*overhead* di memoria introdotto nella rappresentazione dell'oggetto deve essere il minimo possibile.

Dal momento che si realizza il *Reference Counting* si associa ad ogni oggetto un **contatore**. Inizialmente posto su 2 byte, la sua occupazione potrebbe essere ridotta sino al singolo byte*. Bisogna inoltre ricordare che il prototipo gira sulla Java Card come una applet: non c'è, nel contesto della gcApplet, una zona di memoria dove possono allocare le altre applet. E' il modello della memoria deve tenere conto di informazioni come la **dimensione** e la **zona** di memoria che occupa. Questo viene fatto introducendo nella struttura dati associata all'oggetto (di nome **Oggetto**, appunto) altri 7 byte. Di questi 7 byte, 2 byte riguardano informazioni utili solo al debug. La gcApplet, infatti, dà un nome ad ogni oggetto solo per poter registrare le azioni che ha intrapreso e per poterle poi comunicare a chi voglia fare debug. Tale nome risulterebbe inutile in una ipotetica versione distribuita.

Diciamo dunque che ragionevolemente un modello di un oggetto occuperebbe, nella versione definitiva, soli 7 bytes.

L'overhead di spazio, in realtà, non è introdotto solo dal contatore dell'oggetto (risulta ben poca cosa) bensì da una struttura dati aggiuntiva. Lo scopo di quest'ultima è, similmente ai vettori *markbits* e *allocbits* del paragrafo precedente, chiarire quali oggetti siano allocati, siano vivi, e se contengono riferimenti ad altri oggetti.

La struttura prende il nome di **Tabella dei Riferimenti**, ed è una lista pluri-concatenata di un tipo base che si chiama, appunto, **Riferimento**. Il Riferimento, come vedremo dettagliatamente nel Paragrafo 5.3.2, arriverà ad occupare 11 byte +1 booleano per ogni riferimento, oltre a 4 byte per ogni puntatore che modella le dipendenze strutturali.

Senza scendere ulteriormente nel dettaglio (cosa che verrà ampiamente fatta nei prossimi capitoli) si può dire che questa struttura nasce da diverse necessità:

* Stime statistiche confermano che limitare a 256 il numero di riferimenti in contemporanea ad un oggetto è una scelta ragionevole.

- Deve far capire alla gcApplet il ruolo degli oggetti creati dagli utenti nelle **dipendenze strutturali**. Per dipendenza strutturale si intende una relazione gerarchica di tipo padre-figlio tra due riferimenti.
- Deve fornire il concetto di **visibilità** (*scope*) degli oggetti. La gcApplet non sa quando un oggetto muore per uscita di *scope* a meno che questo non gli venga comunicato dall'utente.
- Deve fungere da mediatore nell'accesso agli oggetti. Un programmatore Java sa, quando alloca memoria, come poter accedere agli oggetto creato, come modificarlo, come crearne degli alias, ma non sa dove tale oggetto risieda. Conosce il **nome** del riferimento ma non la locazione dell'oggetto.
- Deve velocizzare le operazioni che tali utenti sono costretti ad inserire nelle loro applet quando creano o modificano oggetti. E' infatti una prerogativa quella di non appesantire troppo il lavoro degli utenti.
- Deve rendere più "astuto" il *Reference Count* nelle fasi di creazione e modifica degli oggetti. Semplifica il compito della Garbage Detection al semplice scorrimento della memoria in cerca di oggetti con contatori nulli[*].
- Ovvia alla lacuna dell'algoritmo *Reference Count* di non saper identificare i cicli (cfr Paragrafo 2.2.2).

La *Tabella dei Riferimenti*, inoltre, fornisce informazioni di *tracing* ad un algoritmo che di *tracing* non è. Si dà una realizzazione fisica separata al riferimento e all'oggetto riferito. E' una scelta quasi obbligata che nasce dal fatto di non poter avere informazioni a tempo di compilazione degli oggetti creati: si supplisce ottenendole a *run-time* con chiamate esplicite.

La gcApplet deve modellare anche il *firewall* del sistema Java Card che isola ogni applet in un suo contesto (cfr Paragrafo 3.9). Tale separazione viene simulata nel modello introducendo delle strutture dati chiamate *Contesti di Gruppo* (**groupContext**) le quali confinano ogni oggetto nell'ambito in cui è definito, viene dato così un senso al modello di memoria creato. E' presente dunque un contesto per ogni package che risulta identificato dal suo *Application Identifier* (**package AID**). Tale struttura dati contiene gli spazi di memoria destinati agli oggetti, la *Tabella dei Riferimenti*, ed altre informazioni utili per il debug e che saranno dettagliate nel Paragrafo 5.3.3. Il *groupContext* occupa 23 byte per ogni contesto.

[*] Non ci saranno le cancellazioni a cascata tipiche del *Reference Counting* (cfr Paragrafo 2.2.2).

Quanto al debug è stata prevista una memorizzazione delle **azioni** intraprese dal Garbage Collector in un **file** di **Log** con lo scopo di poter fornire una documentazione di ciò che accade all'interno della Java Card. Se la gcApplet non le memorizzasse, tali informazioni sarebbero inaccessibili. Si ricorda, infatti, che la Java Card comunica con l'esterno attraverso APDU in modalità *master/slave*. Non ci sarebbe dunque maniera di sapere cosa accada all'interno se non venisse espressamente interrogata. Nel file sono memorizzati informazioni come gli oggetti creati, quelli che sono modificati da un aumento del contatore, la nascita di riferimenti degli oggetti ad altri e molte altre che vedremo più in dettaglio nei Paragrafo 5.3.4 e 5.3.5.

La figura 4.4 è illustra un modello schematico della memoria della Java Card. E' evidenziato l'isolamento delle applet e la loro interazione con la gcApplet mediante lo *SIO*. Le frecce chiariscono come la gcApplet interpreti queste chiamate andando ad operare sul modello della memoria che si è costruito. Nella figura sono evidenziati la presenza, all'interno del modello, dei *Contesti*, degli *Oggetti* e della *Tabella dei Riferimenti*. La gcApplet è anche in collegamento con il *file di Log*, poiché registra in tale archivio tutte gli eventi che accadono durante la sua esecuzione.

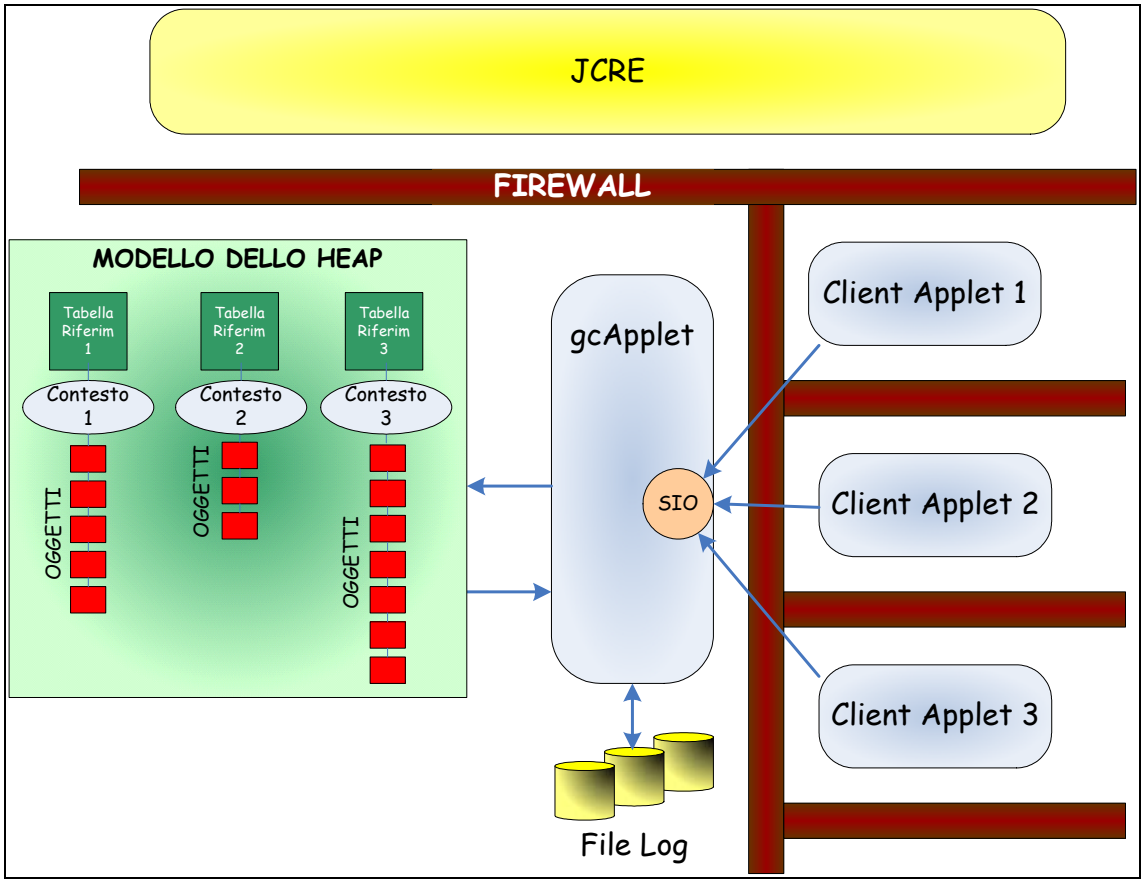

Figura 4. 4 Modello schematico della gcApplet

4.3.2 Funzionamento

Le varie applet presenti sulla carta comunicano con la gcApplet per mezzo del *SIO* messo in condivisione e anzitutto si installano dando informazioni di sé. Questo scatena due eventi:

1. La creazione all'interno del modello di memoria di un *Contesto* con una *Tabella dei Riferimenti* vuota e nessun *Oggetto* (si suppone che all'inizio non ce ne siano).
2. La registrazione nel *file Log* della nascita di un nuovo *Contesto*.

Alla creazione di ogni oggetto sono compiuti tre eventi:

1. La creazione dell'*Oggetto* nel *Contesto* riservato all'applet all'interno del modello di memoria.

2. L'aggiornamento della *Tabella dei Riferimenti* con le informazioni dedotte circa dipendenze strutturali e informazioni di compilazione.
3. Viene registrata una azione nel *file Log* dell'accaduto.

In coincidenza di un assegnamento tra oggetti succede che:

1. Si accede, per mezzo della *Tabella dei Riferimenti* all'*Oggetto* che non sarà più riferito e si decrementa il contatore.
2. Si accede al nuovo *Oggetto* puntato e gli si incrementa il contatore.
3. Si scorre la *Tabella dei Riferimenti* in cerca di eventuali dipendenze tra oggetti per aggiornare il contatore anche di tali oggetti.
4. Si registrano tutte le azioni nel *file di Log*.

Periodicamente oppure quando risulta necessario si può invocare la garbage collection che semplicemente:

1. Scorre gli *Oggetti* in cerca di quelli col contatore a zero.
2. Elimina tali *Oggetti* dalla struttura.
3. Notifica l'eliminazione nel *file Log*.

4.3.3 Alcune Considerazioni

Durante il normale funzionamento della Java Card accadono cambi di contesto necessari affinché la gcApplet svolga il proprio compito. E' un overhead di tempo non trascurabile, ma che potrebbe essere notevolmente abbattuto se tale algoritmo fosse implementato nella Java Card Virtual Machine come dovrebbe.

Il punto di forza della variante dell'algoritmo risiede nell'aver reso immediata la Garbage Detection avendo spostato le computazioni di Garbage Collection al momento della creazione e modifica oggetti anziché alla loro cancellazione. Quando un oggetto è stato trovato essere "garbage", infatti, non bisogna scorrere gli oggetti da esso riferiti per decrementare contatori (cancellazione a cascata) poiché questa è una operazione che è già stata fatta al momento in cui il contatore dell'oggetto è stato decrementato. Le cancellazioni a cascata, dunque, sono state rimpiazzate da un più efficiente decremento (e incremento) a cascata. Questo espediente sarebbe stato impossibile in un algoritmo di *Reference Counting* tradizionale, ed è per questo che la gcApplet è considerata una sua modifica più "astuta".

Sebbene ne si implementi una variante, la scelta non è ricaduta sugli algoritmi di *tracing* poiché questi hanno peggiori rendimenti con l'aumentare delle dimensioni dello heap. Inoltre come temporizzazioni gli algoritmi di *tracing* sono vincolati a dover rendere seriale perlomeno la prima esplorazione. "Fermare il mondo" per scansionare la memoria a partire dal root-set (*mark and sweep*) o anche solo per esplorare la generazione più giovane (*generazionali*), può impiegare un tempo relativamente lungo. L'utente di Java Card si vedrebbe costretto ad una attesa con la carta inserita nel CAD più del dovuto. Nel caso più stringente di una *contactless* card (cfr Paragrafo 3.2), si potrebbe anche oltrepassare il tempo in cui la carta è nel raggio di azione lasciando incompleta la collezione e inconsistente il modello. Le carte senza contatti, infatti, hanno dei requisiti temporali più stretti che renderebbero necessaria la presenza di un Garbage Collector di tipo incrementale.

Le basi gettate nella creazione delle strutture dati sono state indirizzate, come abbiamo visto, dalla scelta del *Reference Count*, ma poche dovrebbero essere le modifiche per passare ad un Garbage Collector di tipo *tracing*. Basterebbe infatti sfruttare la *Tabella dei Riferimenti* e aggiungere il concetto di appartenenza ad un root-set per ottenere subito un criterio di raggiungibilità sfruttabile da algoritmi di tipo *tracing* come il *mark and sweep*. Gli algoritmi generazionali sono ritenuti da alcuni produttori di Java Card i più adatti a tale sistema embedded [GemplusOS]. Aggiungendo al *Riferimento* il concetto di appartenenza ad una generazione, assegnando ad ognuna una differente *Tabella dei Riferimenti* per poter operare l'algoritmo che più aggrada per ogni generazione, sarebbe possibile dotare la gcApplet di un Garbage Collector di tipo *generazionale*.

4.4 L'ambiente di sviluppo

I file sogente del prototipo sono realizzati con l'ambiente di sviluppo Java NetBeans della Sun. Del linguaggio Java sono state utilizzate solo le funzionalità supportate per evitare errori in fase di conversione.

Il Java Card Developers Kit, nella versione 2.2.1 [JCKit2.2.1] offre le utilità di conversione dei file ".class" nel file ".CAP" necessario per essere installato sulla carta. Il *converter* processa i file class che appartengono ad un package, verifica che il linguaggio sia conforme alle specifiche delle Java Card [JCSpec2.2.1] e produce in

uscita un file ".CAP", un file di tipo Java Card Assembly (".JCA"), ed un file di export (".EXP"). Si lancia il file "converter.exe" e si passa, tra gli argomenti, l'*AID* dell'applet e del package.

Il Java Card Developers Kit contiene un tool chiamato **Java Card Workstation Development Environment (JCWDE)** che permette l'esecuzione simulata delle applet come se fossero "programmate" nella ROM della Java Card. Nel lanciare l'eseguibile "jcwde.exe" bisogna, infatti, passare l'*AID* di tutte le applet in esso contenute: l'emulatore si metterà in ascolto sulla porta di default 9025 in attesa di APDU.

Il *JCWDE* non è una implementazione della Java Card Virtual Machine (JCVM), ma usa la Virtual Machine del Java (JVM) per emulare il Java Card Runtime Environment (JCRE). Non sono coperte, inoltre, alcune funzionalità del JCRE come l'installazione dei package, le transazioni, il firewall, la cancellazione esplicita degli oggetti, delle applet e dei package.

La comunicazione con la carta si può fare attraverso una utilità di nome **Apdutool** ("apdutool.exe") che, preparate le APDU come numeri esadecimali, separati da spazi e conclusi da un punto e virgola, le invia alla carta in maniera più leggibile (distinguendo byte CLA, INS e gli altri) e mostra le APDU di risposta con le due Status Word. La figura 4.5 mostra un esempio dell'utilizzo di tale programma. Tutti i codici esadecimali delle APDU sono stati inseriti nel file di testo *new_apdu.txt*.

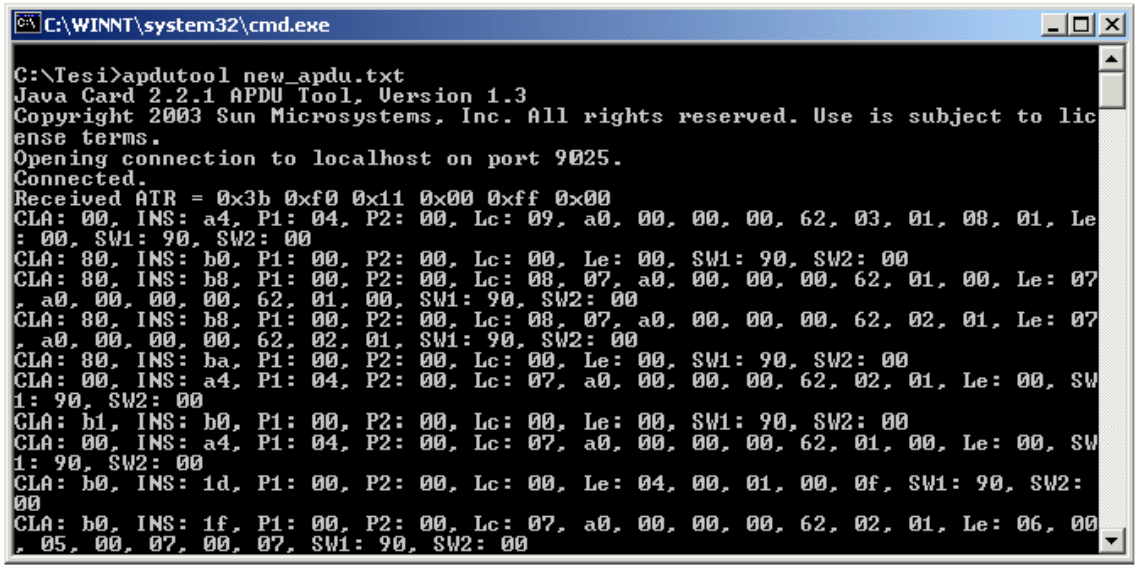

Figura 4. 5 Un Esempio di comunicazione con Apdutool

5. Il Prototipo gcApplet

Nei precedenti capitoli è stato introdotto il contesto in cui va ad operare il prototipo e le scelte progettuali di implementazione. Di seguito viene fatta una panoramica dei componenti di cui è composto il package del prototipo che vengono poi analizzati in maniera più approfondita.

La linea guida è quella di non scendere troppo nei dettagli del codice. Tuttavia, esso sarà riportato nei casi di più stretta necessità, come per illustrare le strutture dati o per fornire esempi chiarificatori. Laddove possa essere utile sono inserite immagini composte nel linguaggio Unified Modeling Language, o UML, che spiegano anche come si relazionano tra di loro le varie strutture dati.

5.1 Il package GC

Il package GC è costituito da sei classi, ognuna atta a descrivere un aspetto del modello di memoria simulato, oppure a svolgere delle funzionalità utili per registrare gli eventi dell'algoritmo di garbage collection.

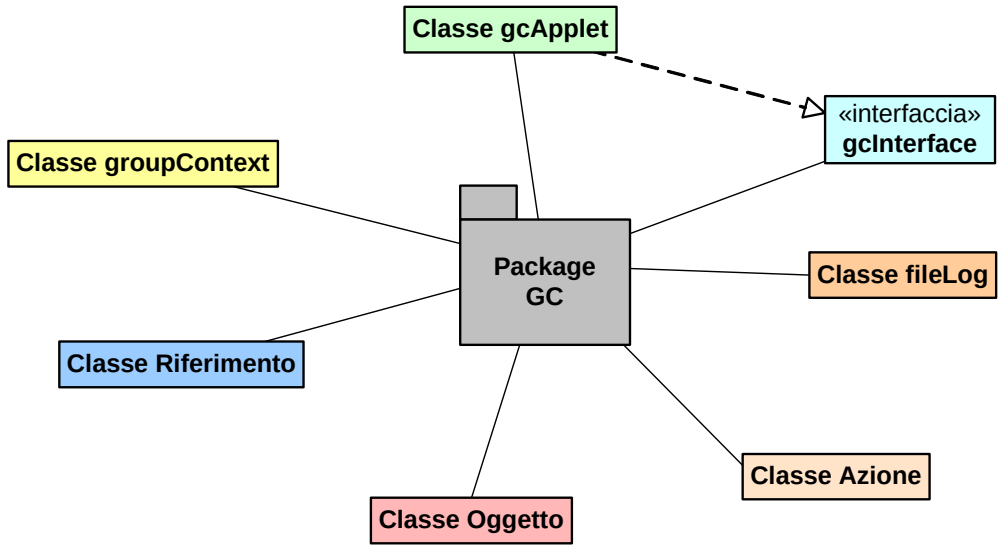

Figura 5. 1 Struttura del package GC

Descrivendo in generale le funzionalità di tale classi si può affermare che:

- **gcApplet** è la classe che coordina le altre. Si avvale delle altre classi per dare un *Contesto* ad ogni applet che si installi. Comanda la creazione di oggetti e riferimenti in maniera congrua alle operazioni che gli vengono notificate. Registra infine ogni azione intrapresa nel *file di Log*.
- **groupContext** è il contesto assegnato ad ogni applet gestita dall'algoritmo. Si occupa di mantenere una lista di *Oggetti* creati ed una di *Riferimenti* (la *Tabella dei Riferimenti*). Ha delle funzionalità di ricerca di elementi nelle liste che vengono sfruttate anche dalla *gcApplet*.
- **Oggetto** è la rappresentazione dei byte allocati in memoria dalle applet. Possiede una variabile *nome* ignota all'applet creatrice che è assegnata dalla *gcApplet* consultando il *groupContext*. Per mezzo di tale nome l'*Oggetto* è acceduto dall'algoritmo di Garbage Collection.
- **Riferimento** contiene il puntatore all'oggetto creato. Costituisce il punto di accesso, stabilito dal programmatore, col quale si può andare a lavorare sugli oggetti in memoria. Ha un proprio tempo di vita e può anche puntare a nessun oggetto, nel qual caso si considera che punti all'oggetto **`null`**.
- **fileLog** è la classe che, istanziata nella *gcApplet*, scrive un file con tutti gli eventi che accadono.
- **Azione** è un semplice file (vettore di byte) che usa delle convenzioni per far capire il tipo e i parametri dell'evento accaduto nel Garbage Collector.

5.2 L'interfaccia gcInterface e manuale d'uso

L'interfaccia costituisce il punto di comunicazione tra la *gcApplet* e le altre applet presenti sulla carta. E' resa condivisibile grazie al meccanismo dello *Shared Interface*: l'oggetto *Shared Interface Object (SIO)* è ritornato da metodo apposito invocato dalle applet (cfr paragrafo 3.9).

I programmatori devono rispettare delle regole di interazione tra le proprie applet e la *gcApplet*.

5.2.1 Vincoli al programmatore

Dovendo la *gcApplet* crearsi un modello di memoria coerente con quello reale e utilizzato dal programmatore, è anzitutto necessario che quest'ultimo rispetti le seguenti regole:

1. Ad ogni oggetto di qualsiasi tipo esso sia (byte, short, una classe ecc.) deve essere dato un nome, univoco all'interno del programma, di tipo short. Con tale *codice short* il programmatore si riferisce all'oggetto quando utilizza i metodi di interfaccia.
2. Di tutti i nomi possibili degli oggetti solo uno è riservato. Tale nome si riferisce al puntatore **`null`** e vale lo short '0x0000';
3. Dovranno essere dati dei nomi a tutti gli oggetti che verranno utilizzati.
4. Dovrà essere inserito nel codice dell'applet una chiamata al metodo di interfaccia nei tempi e nei modi corretti come chiarito dalla loro descrizione nei prossimi paragrafi.

5.2.2 gcInterface

La *gcApplet*, caricata sulla JavaCard, rende disponibile ad ogni altra applet un'interfaccia costituita dalle seguenti variabili e metodi:

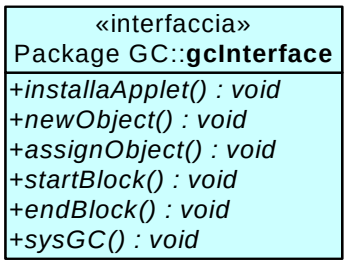

Figura 5. 2 Struttura UML di gcInterface

```
package GC;
import javacard.framework.Shareable;
import javacard.framework.AID;

public interface gcInterface extends Shareable{

     final static short NULL_ID = (short)0x0000;
     public void installaApplet();
     public void newObject(short nome, byte dimensione, short[] figli,
byte startFigli);
     public void assignObject(short nome, short nuovoNome);
     public void startBlock();
     public void endBlock();
   public void sysGC(AID context);
}
```

5.2.3 Manuale d'uso

5.2.3.1 Installazione del package e registrazione con la gcApplet

Anzitutto il programmatore deve ottenere i metodi condivisi dalla *gcApplet* mediante lo *Shared Interface Object (SIO)* seguendo la procedura dettagliata al Paragrafo 3.9.

Negli esempi seguenti si suppone che il programmatore abbia salvato tale *SIO* nella variabile `gc` di tipo `gcInterface` e che faccia tutte le chiamate ai metodi facendoli precedere da `gc` seguito dal punto.

Successivamente risulta necessario per il programmatore notificare l'esistenza del nuovo package alla *gcApplet*, e questo si fa con il metodo:

```
public void installaApplet();
```

E' importante che tale chiamata avvenga prima di intraprendere qualsiasi azione sulla memoria, non devono essere ancora creati o modificati oggetti, quelli già esistenti (inizializzati dal costruttore) dovranno essere notificati subito dopo questa fase con il metodo descritto nel prossimo paragrafo.

La *gcApplet* saprà identificare univocamente il chiamante e comincerà con l'inizializzare le proprie strutture dati atte a simulare lo heap del package.

La scelta di non passare parametri è anche dovuta a una questione di sicurezza: si evita che altri package possano installarsi a nomi falsi oppure doppi con un conseguente invalidamento di tutto il modello costruito dal Garbage Collector.

In fase di progetto sembrava opportuno inserire tale chiamata al momento in cui l'applet effettuava la `install`, in quanto era possibile che già nel costruttore (chiamato al termine della `install`) venissero creati degli oggetti: si sarebbe reso dunque necessaria subito la chiamata ai metodi della gcApplet. Ciò purtroppo è stato impossibile in quanto la chiamata al metodo `install` è effettuata con diritti di Java Card Runtime Environment (diritti di sistema), che rende infattibile la procedura di ottenimento dell'SIO in quanto dovrebbe essere fatta con i diritti utente dell'applet.

5.2.3.2 Nomi delle variabili e NULL_ID

Ad ogni variabile deve corrispondere uno short che ne identifichi univocamente il nome (*codice short* delle variabili).

Ad esempio, se un'applet definisce le variabili:

```
byte [] vett;
short s;
```

allora dovranno essere definite le corrispondenze con i codici short. A titolo di esempio vengono riportate le definizioni di due variabili che forniscono il codice short di entrambi gli oggetti prima creati.

```
final static VETT_ID = (short) 0xC001;
final static S_ID = (short) 0x70A1;
```

Ogni volta che il programmatore utilizza uno dei metodi della *gcInterface*, per riferirsi ad una variabile deve inserirne il *codice short* del nome. Anche se non è necessario memorizzare tale variabile in una costante, farlo renderebbe il codice molto più leggibile.

Infine, la costante resa pubblica dalla *gcInterface* è il codice short per il puntatore **null**. Tale codice short è riservato e, se utilizzato dal programmatore in uno dei metodi di interfaccia, sarà interpretato dalla gcApplet come **null**.

5.2.3.3 Metodo newObject

E' la funzione con cui il programmatore notifica alla *gcApplet* che è stato creato un oggetto di qualche tipo; viene chiamata ogni volta che:

- Si crea un oggetto ti tipo short, byte o boolean.
- Si fa una new.
- Si dichiara solo un oggetto senza però definirlo.

Di seguito è riportata la firma del metodo per una più corretta osservazione.

```
public void newObject(short nome,byte dimensione,short[] figli,byte
startFigli);
```

Gli argomenti passati hanno il seguente significato:

- **nome** è il codice short della variabile che è stata creata. Non può essere `NULL_ID`.
- **dimensione** è il numero di byte che tale variabile occupa. Se uguale a zero significa che la variabile è stata dichiarata ma non definita.
- **figli** è un vettore contenente i codici short delle variabili figlio (vedi più avanti). Può essere **null**.
- **startFigli** è l'indice del vettore `figli` che rappresenta il primo a non essere di tipo byte, short o boolean (vedi più avanti). Quando `figli = null`, `startFigli` può assumere un valore qualsiasi, altrimenti deve essere un indice del vettore dei figli e quindi di valore compreso tra 0 e il numero di figli meno uno.

Segue un esempio per capire i modi ed i tempi di utilizzo di questo metodo.

Partendo dalla definizione delle due variabili dell'esempio precedente il programmatore potrebbe scrivere:

```
byte[] vett = new byte[15];
```

Il *codice short* della variabile è noto (supponiamo sia stato salvato nella costante `VETT_ID`) e la dimensione che esso occupa è 15 bytes. Come sarà chiaro più avanti `vett` non ha figli. La chiamata la metodo newObject che segue è inserita dal programmatore e deve essere così fatta:

```
gc.newObject (VETT_ID,(byte)15,null,(byte)0);
```

Alla stessa maniera una la definizione di 's' dovrebbe essere così seguita (gli short occupano 2 bytes):

```
short s = (short)0xFA11;
gc.newObject (S_ID,(byte)2,null,(byte)0);
```

Se avessi dichiarato la variabile `vett` senza definirla e poi lo avessi fatto in seguito, allora le chiamate ai metodi della gcInterface dovrebbero essere state fatte in questo modo:

```
byte[] vett;
gc.newObject (VETT_ID,(byte)0,null,(byte)0);

short s;
gc.newObject (S_ID,(byte)2,null,(byte)0);

vett = new byte[15];
gc.newObject (VETT_ID,(byte)15,null,(byte)0);

s = (short)10;
```

La prima definizione di vett è, in realtà, una sola dichiarazione, ma deve comunque essere seguita dalla chiamata alla newObject. Dal momento che non viene creato alcun oggetto metto **dimensione** a zero. La stessa cosa non si fa con la variabile s poiché viene creato comunque, per i tipi short così come i byte e i boolean, un oggetto in memoria.

Quando poi definisco vett allora faccio la chiamata alla newObject con la **dimensione** appropriata.

5.2.3.4 I 'Figli' delle variabili

Una variabile complessa, come ad esempio una classe definita da utente, può contenere all'interno variabili di tipo puntatore e avere delle **dipendenze strutturali**, come dimostra il seguente esempio:

```
private class miaClass {
        byte b;
        short s;
        byte [] vb;
        short[] vs;
        miaClass m;
        tuaClass t;
//seguono i costruttori, metodi, ecc..
}
```

Le variabili dichiarate all'interno della classe sono i suoi 'figli'. Esattamente come per le altre variabili, il programmatore deve, ogni volta che crea un'istanza della classe, dare dei nomi anche ai figli creando un vettore di *codici short*. Nel vettore è importante l'ordine dei figli, infatti:

1. Il numero di elementi del vettore deve essere pari al numero di figli inclusi dalla classe.
2. Ad ogni entrata del vettore corrisponde il *codice short* della variabile figlio.
3. Ogni istanza creata della classe deve rispettare lo stesso ordine ma i codici short dei figli devono essere unici nel programma (quindi se coesistono due istanze della stessa classe allora i nomi dei figli devono essere diversi). E' buona norma dare sempre dei nomi diversi ad istanze diverse indipendentemente dai loro tempi di vita.

Sarà dunque possibile tradurre la creazione di un oggetto di tipo miaClass nella seguente chiamata:

```
final static short C_ID = (short)0x0C01;
final static short[] figliC = { (byte)0x0C02, (byte)0x0C03, (byte)0x0C04,
(byte)0x0C05, (byte)0x0C06, (byte)0x0C07};

miaClass c = new miaClass();
gc.newObject(C_ID, (byte)19, figliC, (byte)2);
```

Si fa notare che **dimensione** è pari a 19 poiché c'è un byte, uno short (2byte) e 4 puntatori (4 byte a puntatore). **startF** è pari a 2 perché dal secondo figlio in poi del vettore `figliC` sono tutti puntatori (non sono dei tipi base byte, short o boolean): in questa maniera si fa capire alla gcApplet che, quando ci sarà un assegnamento di due classi di tipo miaClass, solo i primi due figli (figli[0] e figli[1]) dovranno essere assegnati per valore, mentre i restanti saranno assegnati per riferimento variandone l'oggetto riferito.

Un caso particolare è quando si dichiara solamente una classe senza definirla. In tal caso **dimensione** dovrà essere zero, e i figli potranno essere inseriti oppure no, l'importante è che essi vengano passati nella newObject dove la classe è anche definita.

Vediamolo tale caso in termini di codice:

```
miaClass c;
gc.newObject(C_ID, (byte)0, figliC, (byte)2);
// oppure
//gc.newObject(C_ID, (byte)0, null, (byte)2);
```

Viene dato per scontato che eventuali assegnamenti ai membri della classe fatti dal costruttore andranno notificati alla gcApplet dopo che è stata chiamata la newObject.

Quindi se il costruttore della classe `miaClass` è del tipo:

```
public miaclass(byte b, short s, byte [] vb){
      this.b = b;
      this.s = s;
      this.vb = vb;
      this.vs = new short [3];
}
```

Allora il seguente codice del programma sarebbe così tradotto:

```
miaClass c = new miaClass((byte)5, (short)7, new byte[4]);
gc.newObject(C_ID, (byte)19, figliC, (byte)2);
gc.newObject(figliC[2], (byte)4, null, (byte)0);
gc.newObject(figliC[3], (byte)6, null, (byte)0);
```

5.2.3.5 Metodo assignObject

Deve essere chiamato dal programmatore ogni volta che l'assegnamento di due variabili produce un cambiamento dell'oggetto puntato tra i rispettivi riferimenti.

La firma è la seguente:

```
public void assignObject(short nome, short nuovoNome);
```

- **nome** è il codice short della variabile che è assegnata e di cui è modificato il puntatore. Ovviamente non può essere **`null`**.
- **nuovoNome** è il *codice short* del riferimento a cui si viene assegnati. Nel caso di un assegnamento a **`null`** esso deve avere il codice short `NULL_ID`.

Come è noto l'assegnamento di due variabili produce un cambiamento del riferimento solo se queste si trovano nello heap (sono state creati oggetti con l'operatore **`new`**), altrimenti verrà solo copiato il valore della variabile.

Sempre rifacendosi alle variabili degli esempi precedenti, un assegnamento del tipo:

```
short s = (short)5;
short q;
q = s;
```

Produce solo chiamate alla newObject e non ad assignObject. Mentre un assegnamento del tipo

```
byte[] vett = new byte[13];
byte[] p = new byte[5];
p = vett;
```

Deve essere intervallato da chiamata ai metodi di gcInterface in questa maniera:

```
byte[] vett = new byte[13];
gc.newObject (VETT_ID, (byte)13, null, (byte)0);

byte[] p = new byte[5];
gc.newObject (P_ID, (byte)5, null, (byte)0);

p = vett;
gc.assignObject(P_ID, VETT_ID);
```

Inoltre un assegnamento a **`null`** sarà così trattato:

```
p = null;
gc.assignObject(P_ID, NULL_ID);
```

Non sono esenti da tali chiamate neppure gli assegnamenti tra tipi complessi, come le classi, che hanno dei figli. Si veda l'estratto di codice seguente che si rifà alla classe `miaClass` precedentemente definita.

```
miaClass c = new miaClass();
gc.newObject(C_ID, (byte)19, figliC, (byte)2);

miaClass d = c;
gc.newObject(D_ID, (byte)19, figliD, (byte)2);
gc.assignObject(D_ID, C_ID);
```

Con il vettore di short `figliD` avente lo stesso numero di elementi di `figliC` ma entrate diverse.

In pratica, con il passaggio dei vettori figlio, alla *gcApplet* si fa capire quali siano le dipendenze strutturali interne alle classi, in maniera che il programmatore non debba preoccuparsi di esaurire tutti gli assegnamenti tra figli con tante chiamate alla assignObject. Senza il passaggio dei figli, infatti, il precedente assegnamento avrebbe prodotto le successive chiamate come di seguito illustrato.

```
d = c;
gc.assignObject(D_ID, C_ID);
gc.assignObject(figliD[2], figliC[2]);
gc.assignObject(figliD[3], figliC[3]);
gc.assignObject(figliD[4], figliC[4]);
gc.assignObject(figliD[5], figliC[5]);
```

Per di più se per caso il terzo figlio di c puntasse ad un altro oggetto di tipo miaClass, ovvero se c.m != **null**, allora il programmatore dovrebbe inserire anche l'assegnamento dei figli di c.m con quelli di d.m, magari non sapendo neppure, a tempo di compilazione, l'esistenza di tali figli!

Per come è stata realizzata la assignObject, invece, essa scorre tutti i figli dei figli delle classi e ne fa l'assegnamento in maniera corretta.

L'unico vincolo che si chiede al programmatore, già detto ma qui risulta più chiaro, è che inizializzi tutti i riferimenti che usa. In questo caso deve inizializzare anche i figli dei figli se questi gli serviranno. Per tornare all'esempio supponiamo che siano definiti anche i figli di c.m nella seguente maniera, così come quelli di d e dei figli di d.m:

```
final static short C_ID = (short)0x0C01;
final static short[] figliC = { (byte)0x0C02, (byte)0x0C03, (byte)0x0C04,
            (byte)0x0C05, (byte)0x0C06, (byte)0x0C07};
final static short[] figliCm = { (byte)0x0C08, (byte)0x0C09,
            (byte)0x0C0A, (byte)0x0C0B, (byte)0x0C0C, (byte)0x0C0D};
final static short D_ID = (short)0x1C01;
final static short[] figliD = { (byte)0x1C02, (byte)0x1C03, (byte)0x1C04,
            (byte)0x1C05, (byte)0x1C06, (byte)0x1C07};
final static short[] figliCm = { (byte)0x1C08, (byte)0x1C09,
            (byte)0x1C0A, (byte)0x1C0B, (byte)0x1C0C, (byte)0x1C0D};
```

allora la chiamata alla assignObject è equivalente alle seguenti chiamate:

```
d = c;
gc.assignObject(D_ID, C_ID);
/* equivale a
gc.assignObject(figliD[2], figliC[2]);
gc.assignObject(figliD[3], figliC[3]);
gc.assignObject(figliD[4], figliC[4]);
gc.assignObject(figliD[5], figliC[5]);
gc.assignObject(figliDm[2], figliCm[2]);
gc.assignObject(figliDm[3], figliCm[3]);
gc.assignObject(figliDm[4], figliCm[4]);
gc.assignObject(figliDm[5], figliCm[5]);
*/
```

La assignObject prevede, infatti, una funzione che ricorsivamente scorre tutti i figli fino a che non esaurisce tutte le “generazioni”. E’ inoltre prevista la gestione di eventuali cicli con figli che puntano di nuovo a padri, cosa che contraddistingue questo algoritmo da quello tradizionale di *Reference Counting*.

5.2.3.6 Metodo startBlock ed endBlock

Le variabili utilizzate dal programmatore nei propri programmi seguono un tempo di vita ben preciso, e che va da dove vengono dichiarati fino alla fine del blocco in questione.

Di tale situazione il Garbage Collector deve prenderne atto, ed è per questo che si rende necessario per il programmatore notificare alla *gcApplet* quando inizia un blocco e quando poi finisce.

I due metodi sono i seguenti:

```
    public void startBlock();
    public void endBlock();
```

Non servono argomenti in quanto la *gcApplet* identifica automaticamente il numero di blocco ed eventuali blocchi annidati. La *gcApplet* fa fronte anche alle commutazioni di contesto tenendo conto di ogni contesto in maniera separata dagli altri.

Seppure sarebbe teoricamente corretto annunciare ogni inizio blocco ed ogni sua fine, in realtà basta che questo avvenga solo per quei blocchi dove vengono definite nuove variabili.

Ad esempio viene di seguito illustrato il seguente codice.

```
miaClass c = new miaClass();
short s = c.s;
if (c.s != (short)0){
      byte[] vett = new byte[13];
      c.vb = vett
}
else{
      c.vb = null;
}
```

Le chiamate ai metodi della *gcInterface* dovranno essere siffatte:

```
gc.startBlock();
miaClass c = new miaClass();
gc.newObject(C_ID, (byte)19, figliC, (byte)2);
short s = c.s;
gc.newObject (S_ID, (byte)0, null, (byte)0);
gc.assignObject(S_ID, figliC[1]);
if (c.s != (short)0){
      gc.startBlock();
      byte[] vett = new byte[13];
      gc.newObject (VETT_ID, (byte)13, null, (byte)0);
      c.vb = vett;
      gc.assignObject(figliC[2], VETT_ID);
      gc.endBlock();
}
else{
      c.vb = null;
      gc.assignObject(figliC[2], NULL_ID)
}
gc.endBlock();
```

Nel blocco `else` è inutile ma non scorretto chiamare i due metodi startBlock ed endBlock in quanto non viene creato alcun oggetto.

5.2.3.7 Metodo sysGC

Ciò che implementa la simulazione della distruzione degli oggetti non più riferiti è il metodo:

```
public void sysGC(AID context);
```

Nel Java gli oggetti non più riferiti sono deallocati, durante l'esecuzione, in momenti ignoti al programmatore, a meno che questi non effettui la chiamata esplicita alla funzione **System.gc()** del package `java.lang.System`.

Per cercare di ricreare le stesse condizioni il metodo della *gcInterface* può essere usato in due maniere:

1. Se si indica un contesto allora la collezione è effettuata solo nel contesto specificato.
2. Se non si passano argomenti (**`null`**), allora la Garbage Collection è effettuata all'interno di tutto lo heap.

5.3 La gcApplet: strutture dati e metodi

La *gcApplet* è anzitutto una applet: le specifiche del Java Card [JCSpec2.2.1] prevedono che essa derivi dalla classe base **Applet** che appartiene al package 'javacard.framework'. Lo spaccato di codice seguente mostra come ciò sia possibile:

```
package GC;
import javacard.framework.*;
import GC.gcInterface;

public class gcApplet extends Applet implements gcInterface {
```

Di tale classe la *gcApplet* eredita i metodi necessari alla corretta installazione sulla Javacard. Il metodo **`install`** fornisce al Runtime Environment l'*Application Identifier (AID)*. Tra gli altri metodi della classe *Applet* che sono soggetti a overriding da parte della gcApplet c'è il metodo **`process`**, grazie al quale è possibile interpretare correttamente i comandi inviati da esterno sotto forma di Application Protocol Data Unit (APDU). E' presente poi il metodo **`select`**, invocato dal sistema quando l'APDU ricevuta è una di selezione. Infine viene chiamata la **`deselect`** invece quando arriva una APDu di selezione di una nuova applet (cfr Paragrafo 3.7). La funzione **`getShareableInterfaceObject`** permette alla *gcApplet* di condividere la propria interfaccia con le altre Applet, rendendo così disponibili i metodi della *gcInterface* (cfr Paragrafo 3.9).

Viene di seguito elencata la lista dei metodi ereditati, implementati o sovrascritti dalla classe base `Applet` appartenente al package `javacard.framework`.

```
public static void install (byte[] bArray, short bOffset, byte bLenght){
public void process(javacard.framework.APDU apdu){
public boolean select(){
public void deselect(){
public Shareable getShareableInterfaceObject(AID client_aid, byte
parameter)
```

Saranno descritti i metodi di cui si avvale la gcApplet solo dopo aver elencato le strutture dati.

5.3.1 Classe Oggetto

E' la classe utilizzata per rappresentare l'oggetto creato dal programmatore quando alloca memoria. Deve possedere una struttura molto semplice in quanto ne verrà creata una per ogni oggetto esistente in memoria. I requisiti di occupazione di memoria stringenti sono soddisfatti dalla sua scarsa occupazione di memoria (occupa 5 bytes + 1 puntatore, ovvero 9 bytes).

La classe è così composta:

```
public class Oggetto {
    short nome;
    short count;
    byte dimensione;
    Oggetto pun;
```

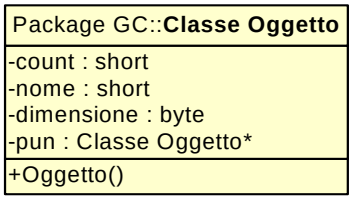

Figura 5. 3 Struttura UML della classe Oggetto

Le variabili definite all'interno assolvono ai seguenti scopi:

- **nome** è l'identificatore dell'oggetto. E' assegnato dalla gcApplet e non è noto al programmatore dell'applet utente. E' garantito essere unico all'interno del contesto in quanto è un valore preso da un contatore, presente nella classe *groupContext*, che è aumentato ad nuova istanza della classe *Oggetto*. Risulta dunque possibile creare fino a 64K oggetti prima che esso perda di consistenza.
- **count** è il contatore dei riferimenti che puntano a tale oggetto. E' la base su cui si sviluppa l'algoritmo di *Reference Counting*. Viene incrementato ogni volta che un c'è un riferimento in più a puntare all'oggetto e viene decrementato ogni volta che un riferimento cambia oggetto puntato (magari anche solo perché finisce il suo tempo di vita). C'è dunque un limite superiore di numero di riferimenti assegnabili ad un oggetto che è pari a 64K.
- **dimensione** è il numero byte che tale oggetto occupa in memoria. Non può essere nullo* e si possono creare oggetti di dimensioni fino a 256 byte.

* Se è una allocazione di memoria dovrà essere di almeno un byte. Questa variabile non necessariamente corrisponde al valore passato dal metodo newObject precedentemente illustrato.

- **pun** serve ad implementare la lista di oggetti (punta all'oggetto successivo della memoria).

Questa classe non ha metodi ma ha il solo costruttore che inizializza un oggetto in base al nome e alla dimensione passatagli. Sul **count** e **pun** operano, rispettivamente, la *gcApplet* e il *groupContext*.

5.3.2 Classe Riferimento

E' il puntatore all'oggetto in memoria, creato dall'utente con una dichiarazione eventualmente seguita da una definizione. Ha un struttura più complessa della classe Oggetto in quanto deve fornire molte più informazioni.

```
public class Riferimento {
    short rif;
    Oggetto ogg;
    byte istanza;
    Riferimento[] figli;
    boolean dirty;
    Riferimento pun;
```

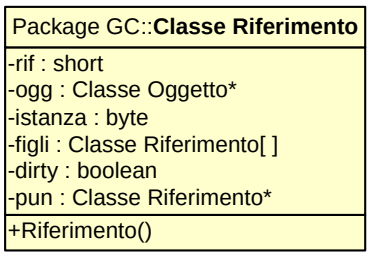

Figura 5. 4 Struttura UML della classe Riferimento

Il significato e l'utilizzo delle variabili è il seguente:

- **rif** è il *codice short* del riferimento. E' stabilito dal programmatore che lo comunica alla *gcApplet* tramite i metodi di interfaccia. Deve essere unico all'interno del contesto, questo limita il numero di riferimenti creabili ad un massimo di 64K.
- **ogg** è il puntatore alla classe *Oggetto*. Quando diverso da **null** indica che il riferimento è stato fatto puntare ad un *Oggetto*. Grazie a questo puntatore è possibile operare l'algoritmo di *Reference Counting* in maniera molto più efficiente e veloce.
- **istanza** è una variabile che serve alla visibilità e tempo di vita del *Riferimento*. Assume un valore dato dal *groupContext* quando il *Riferimento* nasce e rimane tale per tutta la sua esistenza. Serve, alla chiusura di un blocco di istruzioni, a indicare alla *gcApplet* se il *Riferimento* in questione è stato creato in tale blocco e se è, quindi, destinato alla deallocazione poiché uscito di *scope*.

- **figli** è un vettore di *Riferimenti* ognuno puntante al *Riferimento* dichiarato all'interno di quello corrente (cfr Paragrafo 5.2.3.4).
- **dirty** serve ai metodi ricorsivi definiti nella gcApplet per evitare cicli nella ricorsione (è la contromisura ai cicli).
- **pun** è utilizzato per concatenare la lista dei *Riferimenti* puntata dal *Contesto*. Punta all'elemento successivo della lista.

Proprio come la classe *Oggetto*, nella classe *Riferimento* non è presente alcun metodo. Il costruttore della classe si limita ad assegnare il giusto valore passatogli ad ogni variabile interna eccetto la visibilità, che viene invece direttamente assegnata dalla gcApplet.

5.3.3 Classe groupContext

Ne viene creata una per ogni package che si installa con la *gcApplet* e rappresenta il contesto assegnato dal firewall del JCRE in cui è allocata la memoria delle applet (cfr Paragrafo 3.9). La struttura della classe viene qui riportata assieme alla rappresentazione UML della stessa.

```
import javacard.framework.AID;
public class groupContext {
      AID pack_aid;
      Oggetto root;
      short contaOgg;
      short numOgg;
      Riferimento tabRif;
      byte visib;
      short numRif;
      groupContext next;
```

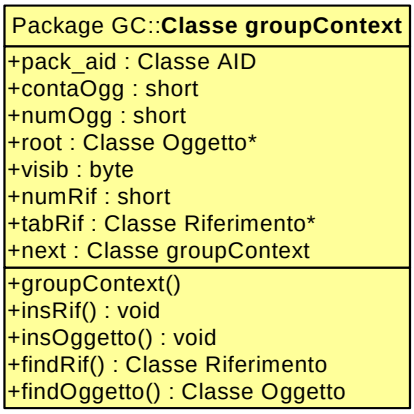

Figura 5. 5 Struttura UML della classe groupContext

Le variabili all'interno della classe significano:

- **pack_aid** è l'identificatore del package installatosi con la *gcApplet*. Essendo una variabile di tipo AID sono sicuro che il nome datogli sarà univoco all'interno della Java Card.
- **root** è il puntatore al primo oggetto della lista *Oggetti*.
- **contaOgg** è un generatore di nomi per gli *Oggetti* utilizzato dalla *gcApplet*: ad ogni creazione incremento il contatore in modo da essere sicuro di non

generare due nomi uguali (questo accaderebbe dopo 64K creazioni). Anche se accadesse il *turn around* del contatore non comprometterebbe il funzionamento dell'algoritmo dato che dare un nome ad un *Oggetto* è utile solo ai fini di debug.

- **numOgg** è il contatore di *Oggetti*, si rende necessario introdurlo per evitare di percorrere tutta la lista per sapere quanti elementi siano presenti. Si riesce a contare fino a 64K *Oggetti*.
- **tabRif** è il puntatore al primo dei *Riferimenti* della lista nota come *Tabella dei Riferimenti*.
- **visib** è il contatore del numero di blocchi di codice aperti ed ancora non chiusi. Indica a tutti gli oggetti in quale blocco sono stati creati per poter poi decrementare il loro contatore quando esso si chiude. Si pone dunque un limite superiore di numero di blocchi annidati pari a 256.
- **numRif** conta il numero di *Riferimenti* nella lista per evitare di farla percorrere.
- **next** è il puntatore al contesto successivo nella lista dei *groupContext*.

Il costruttore semplicemente inizializza le variabili relative al contesto ponendo **pack_aid** pari all'AID del chiamante. Inizialmente *groupContext* risulta non avere alcun *Oggetto* né *Riferimento*: i contatori sono inizializzati a zero e i puntatori alle liste saranno puntatori a **null**.

E' da notare che la installaApplet() non accetta parametri in quanto identifica il chiamante ricavandone l'Application Identifier grazie alla funzione appartenente al `javacard.framework` la cui firma è:

```
public static AID JCSystem.getPreviousContextAID();
```

Questa funzione ritorna l'AID del precedente contesto in esecuzione. Ciò evita che una applet si installi con un identificatore erroneo oppure, caso più grave, deliberatamente voglia farsi scambiare per un'altra applet di un altro package già presente su carta.

I metodi presenti nella classe *groupContext* servono a inserire elementi nelle due liste e a cercare *Oggetti* in base al nome, contatore e dimensione, mentre si possono cercare *Riferimenti* per nome e visibilità.

Sono funzioni di utilità che servono a velocizzare l'algoritmo nelle ricerche e inserimento, non introducono informazioni concettuali.

5.3.4 Classe Azione

Come si evince dalla struttura dati sotto riportata la classe *Azione* è un semplice vettore di byte su cui, seguendo delle convenzioni che saranno poi illustrate, vengono registrate le azioni della *gcApplet*.

```
public class Azione {

    byte [] record
    Azione next;
```

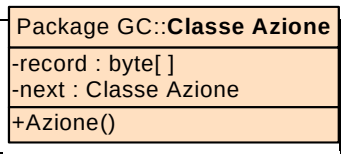

Figura 5. 6 Struttura UML della classe Azione

In questa classe:

- **record** è il puntatore al primo del vettore di byte
- **next** punta alla *Azione* successiva nella lista

Il costruttore, banalmente, inizializza record al vettore passatogli.

5.3.5 Classe fileLog

Istanziata all'interno della classe principale *gcApplet*, la classe *fileLog* gestisce un file in cui registra tutto ciò che accade durante l'esecuzione. Tale file non è altro che una lista di *Azioni*, ognuna contenente i byte registrati.

La struttura viene di seguito riportata:

```
public class fileLog {

    Azione start;
    short numAzioni;
    short numBytes;
```

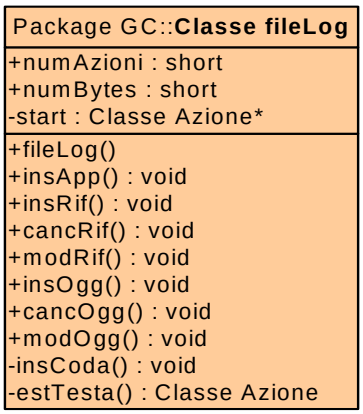

Figura 5. 7 Struttura UML della classe fileLog

Le tre variabili di classe hanno il seguente significato:

- **start** è il puntatore alla prima della lista *Azioni*.
- **numAzioni** evita di scorrere la catena per sapere quante elementi ci siano. Serve soprattutto alle funzioni della *gcApplet* che comunicano con l'esterno per preparare più efficientemente le APDU di risposta.

- **numBytes** è il contatore di quanti byte occuperebbe il file se fosse di tipo sequenziale. Comodo perché i metodi che comunicano all'esterno hanno bisogno di sapere quanti byte vengono inviati per creare le R-APDU.

A parte il costruttore, che semplicemente inizializza la lista ad una lista nulla da zero *Azioni* e zero byte occupati, e a parte due funzioni di utilità come l'inserimento in coda e l'estrazione dalla testa della lista, gli altri metodi implementano lo standard scelto per registrare gli eventi. Viene creata una *Azione* nel cui vettore di byte il primo identifica il tipo di operazione e poi, a seconda dell'operazione, seguono gli altri byte. Ogni *Azione* ha al suo interno, i byte dell'AID del contesto a cui appartiene per rendere unici i nomi degli oggetti e dei riferimenti che vi sono all'interno.

- **InsApp( )** viene chiamata ogni volta che un nuovo package si installa con la *gcApplet*. Al byte di tipo operazione ne segue uno che indica che i successivi (da 6 a 16 bytes) sono di tipo AID.
- **InsRif( )** e **insOgg( )** sono chiamati ogni volta che, nella gcApplet, viene creato un nuovo *Oggetto* o *Riferimento*. Al byte di tipo operazione e all'AID seguono i dati della creazione: ogni campo dell'*Oggetto* (nome, contatore, dimensione) e del *Riferimento* (nome, visibilità), è preceduto da un codice di un byte che ne indica il tipo.
- **cancRif( )** e **cancOgg( )** registrano la cancellazione di un *Riferimento* o di un *Oggetto* semplicemente postponendo al byte di tipo operazione e a quelli di AID, il codice di nome seguito dal nome stesso.
- **modRif( )** e **modOgg( )** sono chiamati alla modifica del *Riferimento* o dell'*Oggetto* puntato e riportano di nuovo tutti i campi della classe come se fosse un inserimento.

5.4 Classe gcApplet

Dopo la panoramica su tutte le classi di cui si avvale, viene adesso esaminata la struttura della classe *gcApplet*:

```
public class gcApplet extends
Applet implements gcInterface {

      groupContext heap;
      short numbytes;
      short numPack;
      fileLog codaAzioni;
```

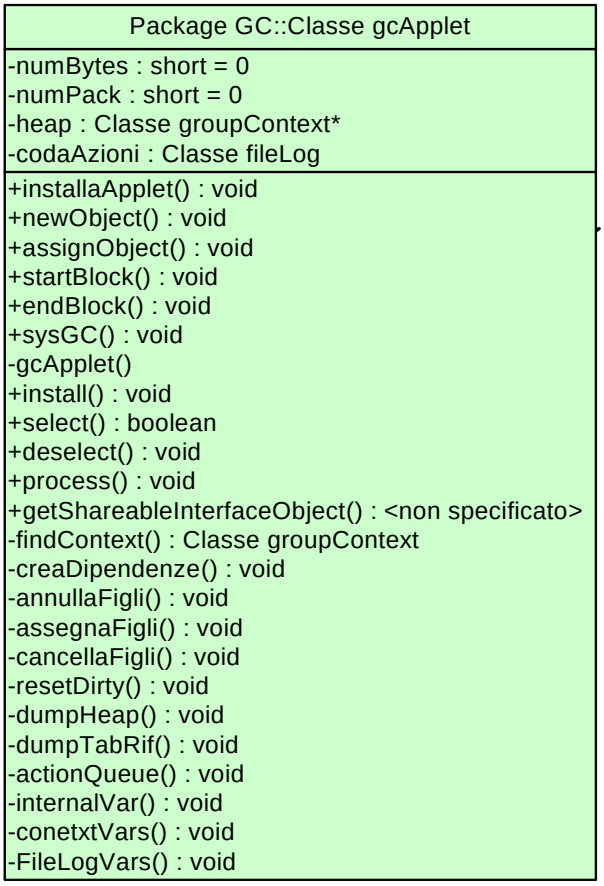

Figura 5. 8 Struttura UML della classe gcApplet

Le variabili di classe sono:

- **heap** che, se diverso da **null** è il puntatore al primo *contesto* della lista. E' usato sempre da ogni metodo di interfaccia per trovare il giusto *contesto* dell'applet chiamante.
- **numbytes** che conta il numero di byte degli *Oggetti* definiti in ogni contesto dalle varie applet. Serve solo per dare un ordine di grandezza delle quantità di dati in gioco. Si riescono a contare fino a 64Kbytes di oggetti.
- **NumPack** è il numero di package distinti che vengono attualmente gestiti dalla *gcApplet*, assume solo scopi di debug.
- **CodaAzioni** è, infine, l'oggetto di tipo *fileLog* su cui vengono chiamate tutte le registrazioni degli avvenimenti del programma.

Oltre ai metodi che appartengono alla gcInterface e a quelli ereditati dalla classe `Applet` del `javacard.framework`, sono presenti altre funzioni di utilità:

- **findContext( )** trova il *contesto* dell'applet di cui si passa l'AID.

- **CreaDipendenze( )**, **annullaFigli( )**, **cancellaFigli( )** e **resetDirty( )** sono funzioni che chiamate al momento opportuno o dalla newObject o dalla assignObject, vanno a esplorare le dipendenze figlio di ogni *Riferimento* modificandoli, cancellandoli o creandone di nuovi. La resetDirty serve solo a resettare il bit di dirty nel *Riferimento* per evitare che l'esplorazione ricorsiva dei metodi suddetti cada in un ciclo.

Seguono i metodi con cui la *gcApplet*, se interrogata, può parlare con l'esterno creando APDU ad hoc per la comunicazione. Sono utili ai soli scopi di debug.

- **dumpHeap( )** fa tornare un elenco degli oggetti presenti nel *contesto* richiesto (che gli viene passato per argomento). E' una o più APDU che hanno come dati una serie di byte elencante il nome, il contatore e la dimensione di ogni *Oggetto* preceduto dal byte di tipo proprio come nel *fileLog*.
- **dumpTabRif( )** ritorna l'elenco dei *Riferimenti* del *contesto* richiesto. E' una serie di nome, nome *Oggetto* puntato e visibilità fatta per ogni riferimento della lista.
- **ActionQueue( )** scorre la codaAzioni e ritorna senza modifiche la serie di azioni che vi sono registrate.
- **InternalVars( ), ContextVars( ) e fileLogvars( )** servono solo a comunicare all'esterno i contatori rispettivamente, dei package e byte gestiti, del numero *Oggetti* e *Riferimenti*, e del numero di *Azioni* e byte occupati.

5.5 Modello riepilogativo

La panoramica delle classi utilizzate nel package GC è evidenziata dal diagramma UML di figura, dove sono evidenziate le dipendenze e le molteplicità tra le strutture.

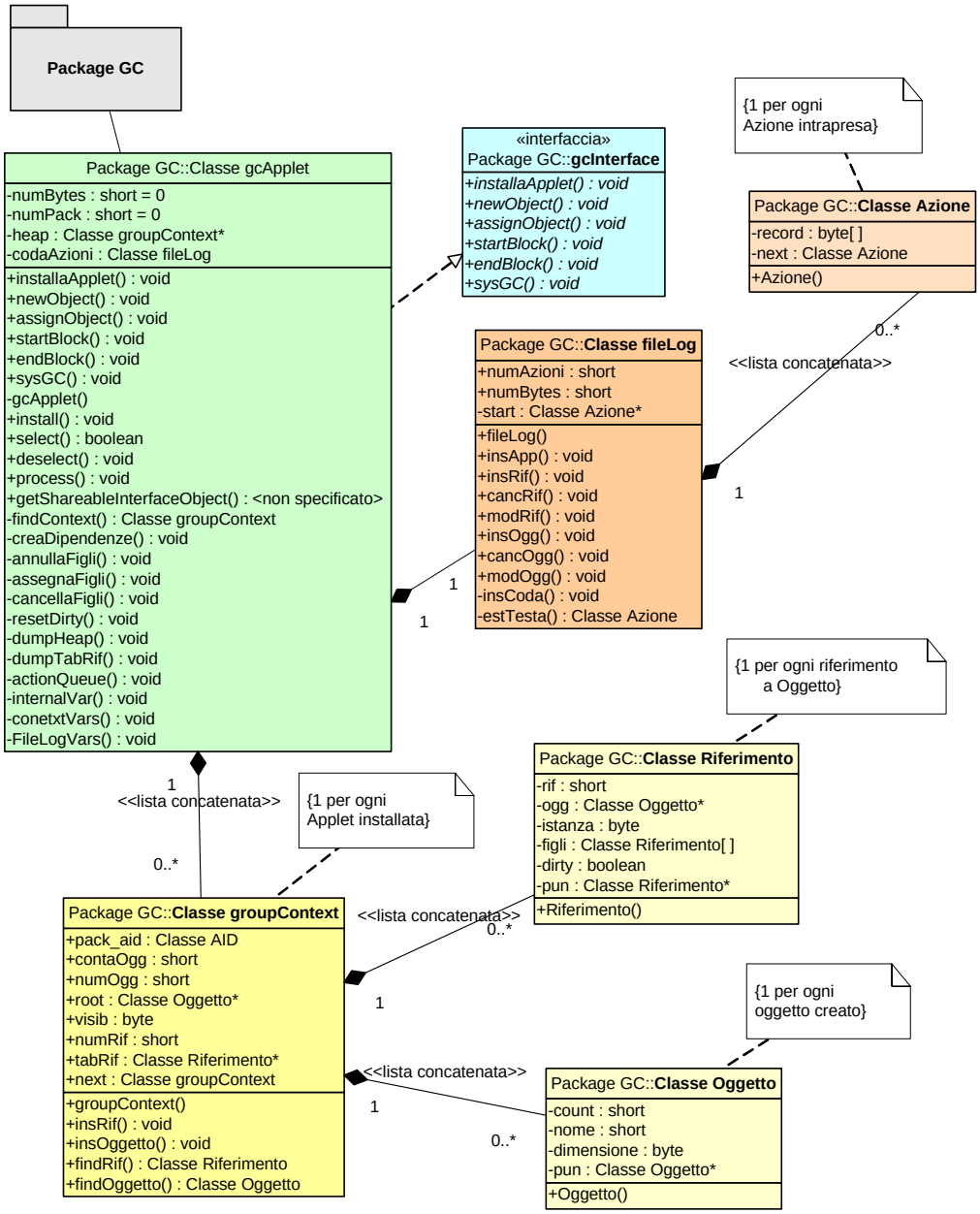

Figura 5. 9 Struttura UML del package GC

6. Un caso d'uso

In questo capitolo viene riportato un esempio di codice di una applet che si installa con la *gcApplet* e che compie delle semplici operazioni di creazione e di modifica di oggetti.

Il codice riportato è commentato e vengono riportate delle illustrazioni, in linguaggio UML, che chiariscono come viene costruito il modello di memoria e quali azioni vengono registrate dalla *gcApplet*.

6.1 Installazione dell'applet client01 e ottenimento dello SIO

Il seguente codice riporta i passi che l'applet *client01* compie per ottenere lo *Shared Interface Object* , in accordo al manuale d'uso di cui al Paragrafo 5.2, supponendo che sappia l'AID della gcApplet e che questa sia memorizzata nella variabile `GC_AID` (linee 20-21).

File Client01.java

```
 7 package Client01;
 8
 9 import javacard.framework.*;
10 import GC.gcInterface;
11
12 public class clientApplet01 extends Applet {
13 //.....
20      final static byte[] GC_AID = new byte[]{(byte)0xA0,(byte)0x00,
21      (byte)0x00, (byte)0x00, (byte)0x62, (byte)0x01,(byte)0x00};
22 //.....
40      private static GC.gcInterface gc;
41 //.....
```

Per essere una Java Card Applet bisogna derivare dalla classe base `Applet` (12) appartenente al `framework javacard`, che quindi va importato (9). E' inoltre preparata una variabile `gc` destinata a contenere lo SIO, che, nel caso della *gcApplet* risulta essere l'interfaccia *gcInterface*.

Fatto questo si passa all'ottenimento vero e proprio dello SIO seguendo i passi descritti al Paragrafo 3.9 e che qui sono inseriti in una funzione apposita di nome `reg()`.

File Client01.java

```
 94      public void reg(){
 95
 96          AID gc_aid = JCSystem.lookupAID(GC_AID, (short) 0,
 97                                          (byte) GC_AID.length);
 98
 99          if(gc_aid == null)
100              ISOException.throwIt(SW_SERVER_AID_NULL);
101          gc = (GC.gcInterface)
102              JCSystem.getAppletShareableInterfaceObject(
103                                              gc_aid, (byte) 0);
104          if(gc == null)
105              ISOException.throwIt(SW_SIO_NULL);
106
107          gc.installaApplet();
108
109      }
```

la `lookupAID` ritorna una variabile di tipo AID, che, essendo un oggetto *entry point* di tipo *permanente* (cfr Paragrafo 3.9), può essere copiato in una variabile locale di nome `gc_aid` (96). Su tale variabile viene fatto un controllo che può sollevare una eccezione di tipo server assente*(100), nel caso l'AID non corrisponda ad alcuno presente nella carta. Se tutto va a buon fine lo SIO viene posto nella variabile `gc` (101-103) e risulterà dunque possibile accedere direttamente ai metodi condivisi dalla *gcApplet* (107).

La chiamata `installaApplet`, a cui non si passano argomenti, provoca un cambio di contesto: si passa ad eseguire il seguente codice della *gcApplet* †.

File gcApplet.java

```
129      public void installaApplet() {
130          //...
132          AID applet_aid = JCSystem.getPreviousContextAID();
133          //...
138          nuovo = new groupContext(applet_aid);
139          //...
141          this.numPack++;
142          codaAzioni.insApp(applet_aid);}
```

* Tale eccezione determina l'interruzione della esecuzione ed un conseguente invio di R-APDU con *trailer* avente come SW1 e SW2 proprio lo short corrispondente a SW_SERVER_AID_NULL.

† Le parti di codice che descrivono l'applet sono già state ampiamente trattate nel precedente capitolo e non verranno qui riportate.

Di particolare interesse risulta la chiamata alla funzione di sistema `getPreviousContextAID()` (linea numero 132) che appartiene al package `javacard.framework.JCSystem`. Questo metodo serve ad ottenere una istanza dell'oggetto AID, di proprietà del JCRE, associata all'applet il cui contesto era precedentemente attivo.

In questa maniera la *gcApplet* può identificare univocamente il chiamante (come era stato specificato in fase di progetto al Paragrafo 5.2.3.1) senza che questi debba identificarsi esplicitamente. Si evitano così errori di riconoscimento voluti e non.

Ottenuto l'AID del chiamante la *gcApplet* crea un nuovo contesto (138), aggiorna un contatore interno di package gestiti, e registra nel *fileLog* che è stata installata l'applet `Client01` identificata dal suo AID (142).

Il modello che descrive le strutture dati può essere schematizzato dalla figura 6.1, nel quale ci sono delle semplificazioni in termini di valore registrato nella *Azione* (dove in realtà c'è un codice che significa la stessa cosa), che qui è riportato in linguaggio naturale. Si dà inoltre un nome al *groupContext* per far capire a quale contesto si riferisca (vedremo poi che questa cosa sarà fatta anche per i nomi degli oggetti e dei riferimenti) e si indica tale nome anche nella variabile pack_AID, che invece dovrebbe contenere esadecimali. Si indica, inoltre, con una freccia tratteggiata congiungente due classi, che all'interno di una c'è un puntatore (il cui nome è riportato sulla freccia), che punta proprio a tale istanza di classe. Nel caso di puntatore a **`null`** la freccia o non è riportata, o avrà il terminale morto (non punterà a niente). La freccia continua, invece, indica che una istanza della classe è contenuta nell'altra (qui la classe *gcApplet* contiene una istanza della classe *fileLog* e che si chiama *codaAzioni*).

Si suppone che, inizialmente, non ci sia alcun package gestito, e che quindi `Client01` sia il primo (`numPack = 1`). Si suppone inoltre che l'AID di tale package stia su 10 byte*, e che quindi l'*Azione* corrispondente "Azione01" ne occupa 12 (2 byte sono di codici interni).

* Si ricorda, dal Paragrafo 3.7, che l'AID va da un minimo di 5 bytes ad un massimo di 16 bytes.

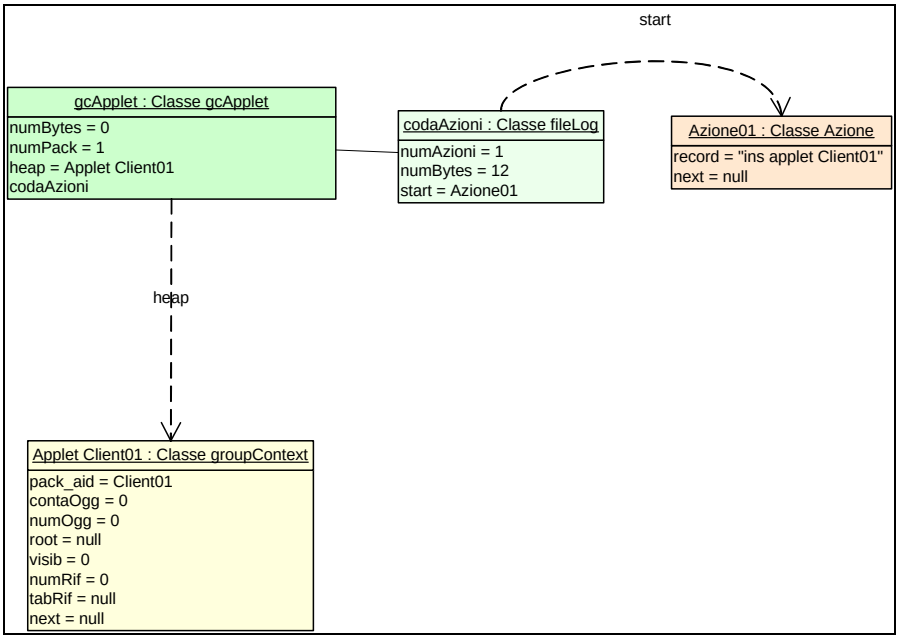

Figura 6. 1 Strutture dati dopo l'installazione dell'applet Client01

6.2 Creazione di un Oggetto

Supponiamo adesso che, durante l'esecuzione dell'applet `Client01`, venga creato un *Oggetto* proprio come nel seguente listato di codice.

File `Client01.java`

```
174          //...
175          final short VETT_ID = (short)0xC765;
176          //...
180          byte[] vett = new byte [2];
181          gc.newObject(VETT_ID, (byte)2, null, (byte)0);
182
183          //...
```

Si è preferito memorizzare il *codice short* del vettore di byte `vett` in una costante di nome `VETT_ID` di valore arbitrario (175) e col quale verranno fatto tutte le chiamate ai metodi della *gcApplet*. In questo esempio viene creato `vett` e, a tale creazione nello heap viene subito notificata (*write-barrier* esplicita, cfr Paragrafo 4.2) alla *gcApplet* con la chiamata al metodo `newObject` (181), informando che tale oggetto, di nome `VETT_ID`, occupa 2 bytes e non contiene puntatori.

Come si vede dalla figura 6.2, la chiamata a tale metodo scatena la creazione di un *Riferimento* "Rif01" il cui nome è indicato con "VETT_ID" (anziché l'illeggibile codice `0XC765`) e il cui campo ogg punta all'*Oggetto* "Ogg01" che, nel modello, rappresenta il vettore di byte a 2 componenti. Infine, tali creazioni saranno documentate nella *codaAzioni*, dove, saranno riportati i valori di tutti i campi di "Ogg01" e di "Rif01".

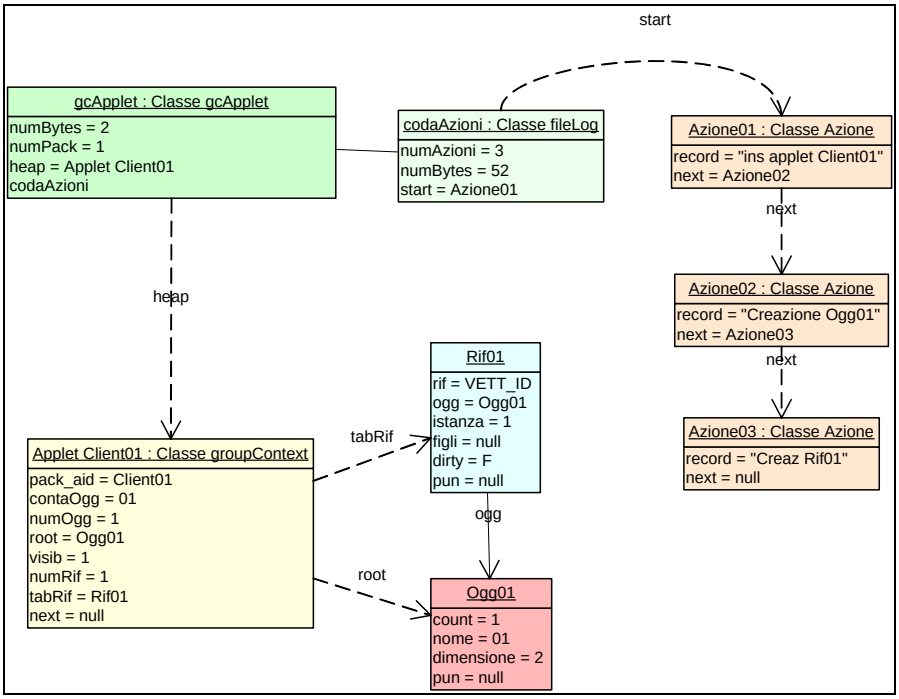

Figura 6. 2 Strutture dati dopo la chiamata alla newObject

Dalla *Tabella dei Riferimenti* risulta che è stato creato un puntatore di nome VETT_ID che punta all'area di memoria corrispondente ai 2 bytes dell'*Oggetto* E' un vettore di bytes a due componenti (ma potrebbe essere anche uno short, la cosa che interessa è che occupi 2 bytes), il cui nome è deciso internamente dalla *gcApplet* mediante un contatore che rende univoco tale nome e che qui vale "01". Come detto in fase di progetto questo capo potrebbe essere omesso, dato che è utile solo al debug perché tale nome viene indicato nella corrispondente "Azione02". Notiamo inoltre il contatore dell'*Oggetto* che vale 1 poiché è riferito da un solo puntatore, e la variabile numBytes, nella classe *gcApplet*, che tiene il conto di tutti gli oggetti creati in tutti i contesti.

6.3 Assegnamento di un Oggetto

Supponiamo adesso che l'applet `Client01` crei un nuovo oggetto proprio come nel seguente spezzone di codice (si faccia riferimento ai numeri di linea):

File Client01.java

```
184         final short ALIAS_ID = (short)0xB716;
185         byte[] alias = vett;
186         gc.newObject(ALIAS_ID, (byte)0, null, (byte)0);
187         gc.assignObject(ALIAS_ID, VETT_ID);
188         //...
```

Come indicato nel manuale d'uso (Paragrafo 5.2.3.5), la definizione per assegnamento deve essere fatta prima creando un riferimento a **`null`**, ovvero una chiamata a newObject con "**`dimensione = 0`**" (186), e poi l'assegnamento vero e proprio (187).

Le figure 6.3 a) e b) mostrano separatamente queste due operazioni. Per esigenze di spazio non sono riportate le azioni registrate nel **fileLog**, poiché simili a quelle dovute alla creazione di `vett`.

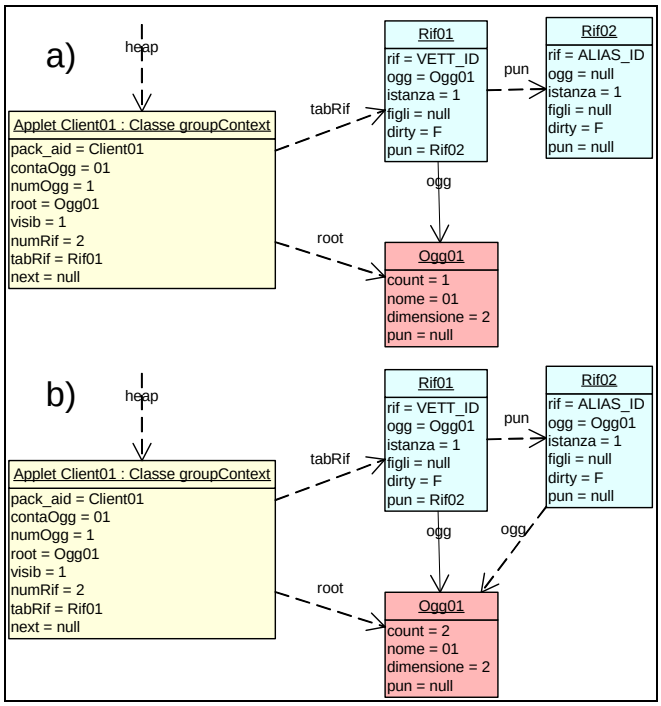

Figura 6. 3 Strutture dati dopo a) la creazione di ALIAS b) l'assegnamento all'oggetto puntato da VETT

Si noti come il fatto di assegnare un nuovo *Riferimento* ad "Ogg1" modifichi esplicitamente il puntatore ogg di "Rif02" da **`null`** a tale *Oggetto*, e faccia aumentare il suo contatore portandolo a valere 2.

6.4 Gestione di più applet

Supponiamo adesso che si installi con la *gcApplet* un'applet di nome `Client02` appartenente ad un nuovo package. I passi che essa seguirà saranno gli stessi visti per `Client01`. Viene di seguito riportato il codice dell'applet che crea più oggetti.

File Client02.java

```
217         //...
218         final short X_ID = (short)0x4658;
219         final short X_S_ID = (short)0x4659;
220         final short X_VB_ID = (short)0x465A;
221         final short X_VS_ID = (short)0x465B;
222         final short[] figliX = {X_S_ID, X_VB_ID, X_VS_ID};
223         final short VETTB_ID = (short)0x465C;
224
225         class cl{
226             short s;
227             byte[] vb;
228             short[] vs;
229         };
230         //...
257         gc.startBlock();
258         cl x = new cl();
259         gc.newObject(X_ID, (byte)10, figliX, (byte)1);
260
261         x.s = (short)5;
262         if(x.s < (short)10){
263             gc.startBlock();
264             byte[] vettb = new byte[8];
265             gc.newObject(VETTB_ID, (byte)8, null, (byte)0);
266
```

Per quanto riguarda la memorizzazione dei *codici short* delle variabili in costanti (218-223) si fa notare la presenza del vettore di short `figliX` (222). Questo vettore memorizza i figli di una istanza della classe `cl` che prende il nome `x` (258). Di tale classe è riportata la struttura (225-229) per chiarire che essa ha tre figli: uno di tipo short, un vettore di byte ed un vettore di short.

All'inizio di un metodo si fa la chiamata alla startBlock (257) che incrementa il contatore dei blocchi annidati chiamato "visib" (presente all'interno della classe `Client02` di tipo *groupContext*). Prima questa chiamata era stata omessa per non complicare troppo l'esempio, adesso invece è riportata. Si suppone inoltre, come vedremo nella figura riassuntiva, che questo blocco sia il primo, e che quindi il contatore valga 1 subito dopo tale chiamata. Quando sarà eseguito il ramo `then` dell'istruzione `if`, ci sarà una nuova chiamata alla startBlock che porterà il contatore "visib" a 2 (dato che la simbolica condizione risulta vera).

Nella chiamata alla newObject di cui alla riga 259, si passa il vettore dei figli di `x` che farà creare tre *Riferimenti*: uno per il padre e puntante all'oggetto classe `cl` che comprende lo spazio occupato dal primo dei tre figli, e due per i figli di tipo puntatore e puntanti inizialmente a **`null`**.

Le strutture dati sono riportate in figura 6.4.

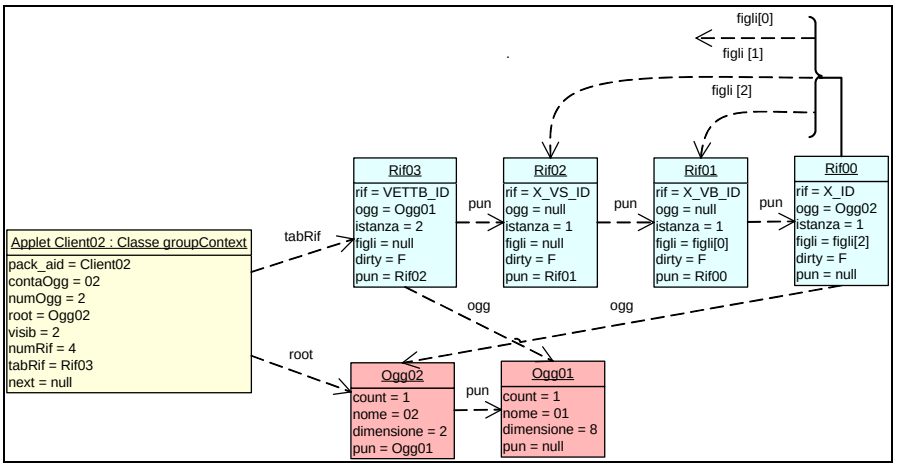

Figura 6. 4 Struttura dati dopo un esempio più complesso

Si esamini il codice qui riportato tenendo presente che è la continuazione di quello precedente:

```
267             //...
269             gc.endBlock();
274         }
```

Alla chiusura del blocco dell'`if`, preceduta da una chiamata della endBlock (269), `vettb` esce di scope. Grazie a tale metodo la gcApplet lo capisce e scorre i riferimenti

in cerca di quelli con "istanza" pari all'ultimo valore della "visibilità" di contesto, cancella i *Riferimenti* dalla tabella e aggiorna gli *Oggetti* puntati.

La situazione che ne consegue è quella riassuntiva di figura 6.5, dove sono omesse le molte azioni registrate nel *file Log* per esigenze di spazio. Si noti come i due *contesti* sono legati l'un l'altro e come questo sia un modello assai intuitivo e realistico di quello che accade nello heap della Java Card.

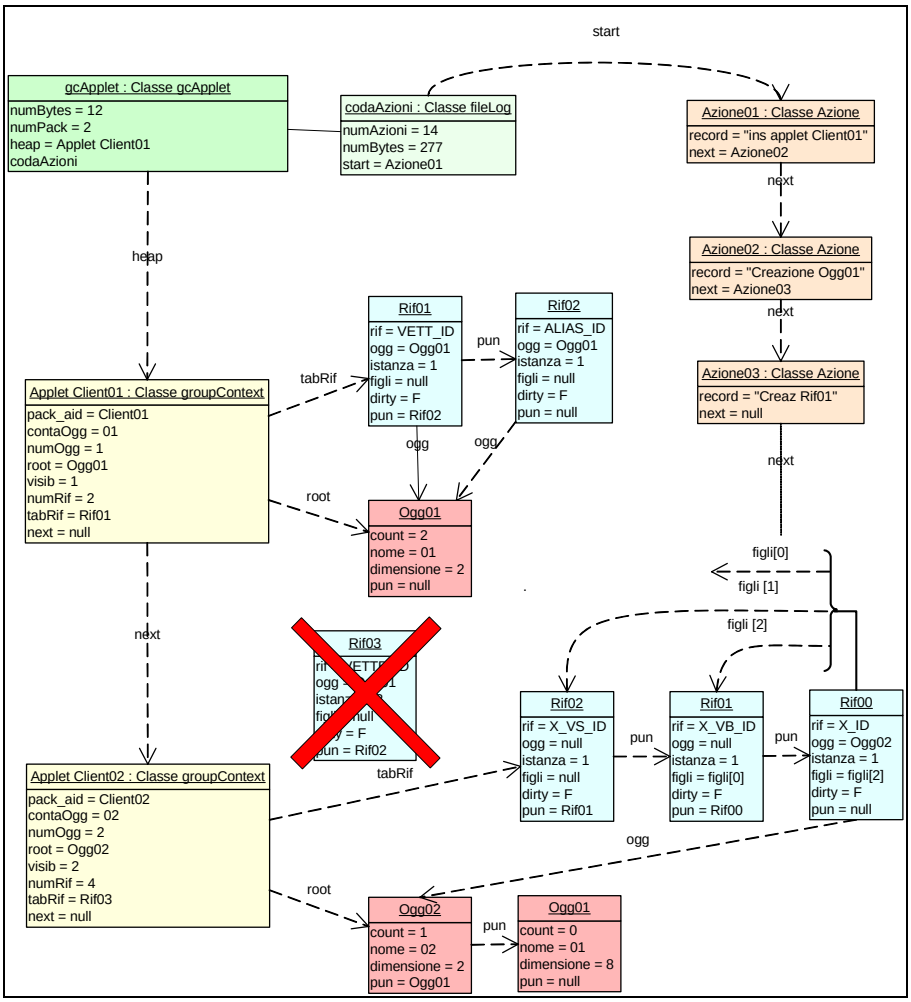

Figura 6. 5 Struttura riepilogativa dopo l'eliminazione del riferimento VETTB

Una invocazione alla funzione di collection `sysGC()` provocherebbe lo scorrimento dello heap e, in ogni *contesto*, della lista *Oggetti* in cerca di quelli con contatore zero.

In questo semplice esempio ce n'è uno, ovvero "Ogg01" del contesto "AppletClient02", che verrebbe rimosso dalla lista e tutto ciò sarebbe anche registrato nella **codaAzioni**.

6.5 Esempio con JCWDE

Di seguito è illustrato l'esempio dell'applet Client01 sotto l'emulatore JCWDE (cfr Paragrafo 4.4).

L'emulatore è lanciato in una finestra separata e resta in ascolto sulla porta 9025.

Figura 6. 6 L'esecuzione dell'emulatore JCWDE

Nel file "*gc.app*" sono presenti gli Application Identifier (AID) dell'*installer*, che permette il caricamento delle applet nella ROM della carta, della *gcApplet* e dell'applet `client01`.

Prima di mostrare quale sia l'output generato da tale esempio occorre specificare alcuni dettagli che rendono più chiara la sua lettura.

Le APDU sono inviate con il comando *apdutool* sotto forma di serie di byte conclusi da un punto e virgola e sono riproposte in una forma più leggibile con specificati i vari campi. Ricordando il significato di questi ultimi (cfr Paragrafo 3.3) si noti come il byte 'Le' è sempre imposto pari a "`0x7F`" (ovvero si richiederebbero sempre 127 byte in ritorno). Non è importante, in quanto il vero numero di byte è imposto dalla applet, e infatti nella forma leggibile in cui è riproposta la variabile 'Le' assume il valore

corretto, seguita dai byte di ritorno. Inoltre il client è supposto avere AID su 7 bytes e di valore arbitrario* pari a "0xA0 0x00 0x00 0x00 0x62 0x02 0x01".

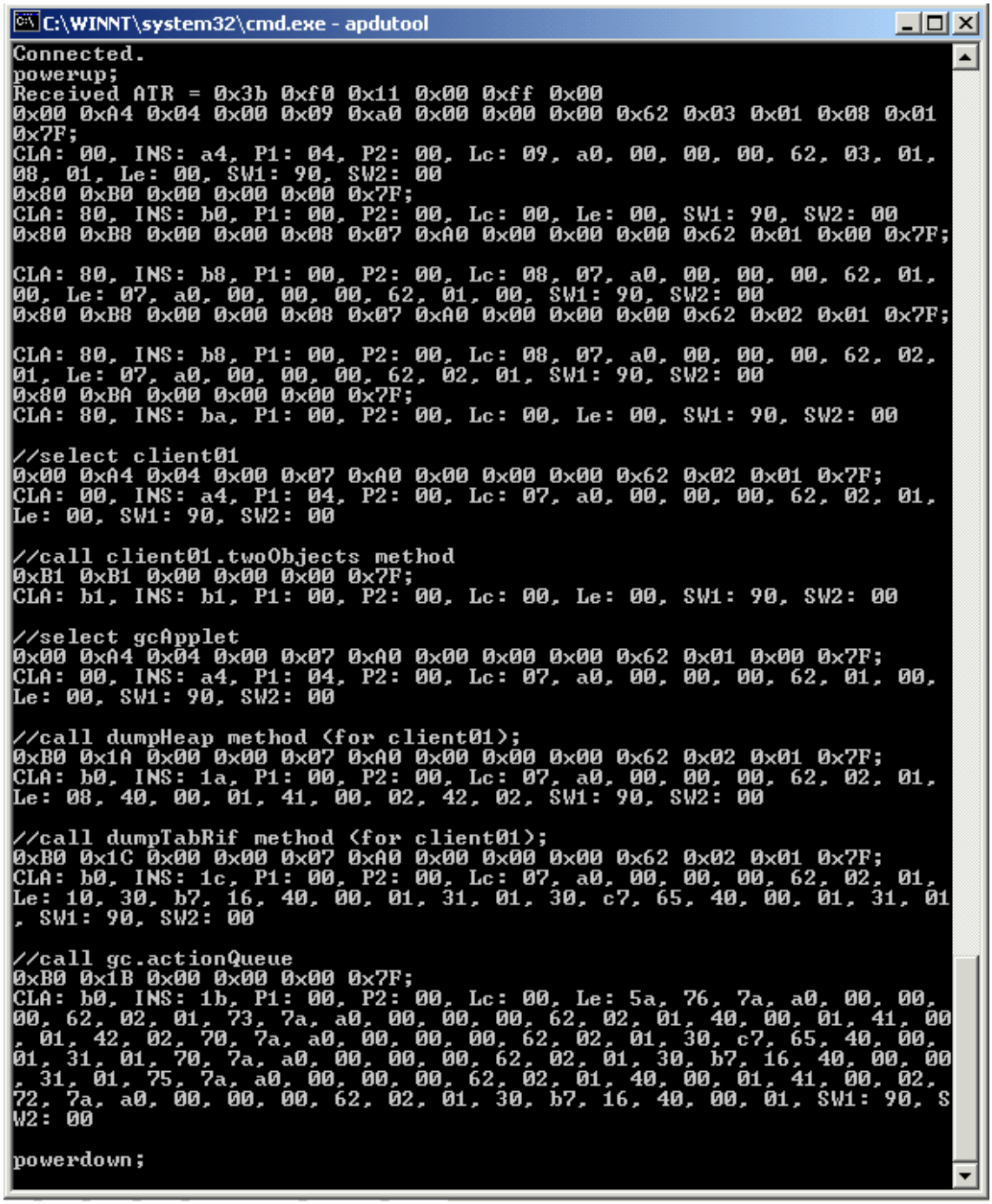

Figura 6. 7 L'output del JCWDE

* In realtà con alcune scelte di AID l'ambiente di simulazione dava degli errori. Nella pratica il problema non si pone poiché è ISO ad assegnare tali identificatori ad ogni produttore.

Analizziamo ora le varie fasi, indicate dai numeri alla sinistra della figura 6.7:

1) La prima parte dell'output della figura 6.7, quella senza commenti, è una procedura necessaria richiesta dall'ambiente per caricare le applet nella ROM. Sostanzialmente consiste nel simulare l'alimentazione della carta con il powerup, nella selezione e partenza dell'*installer*, nella creazione delle istanze delle due applet e nella terminazione dell'*installer*.
2) Anzitutto si seleziona l'applet `Client01`. Come si vede al numero 2, i byte `CLA`,`INS`,`P1` e `P2` assumono un particolare valore prestabilito [JCKit 2.2.1] e che fa capire alla Java Card che l'operazione è di install. Nei dati, che sono 7 bytes come indicato dal primo valore subito dopo `Lc`, è passata l'AID dell'applet `Client01`. In ritorno non si ottengono bytes di dati (`Lc = 0x00`) ma le Status Word valgono `0x90 0x00` stanti ad indicare il successo dell'operazione. E' stato deciso in fase implementativa dell'applet, di porre qui l'ottenimento dello SIO e della successiva installazione con la *gcApplet*.
3) Una volta avvenuta la selezione si procede ad invocare il metodo dell'esempio che crea due riferimenti e li fa puntare allo stesso oggetto. Per poter fotografare la situazione nell'istante prima in cui si chiude il blocco del metodo, dove altrimenti morirebbero entrambi i riferimenti, si è commentata la chiamata alla endBlock, facendo "credere" così alla `gcApplet` che il metodo non sia ancora finito. Come risultato di tutto ciò sono ritornate le SW di successo.
4) Si desidera adesso vedere la situazione del modello della memoria che la gcApplet si è creato, e per far questo bisogna anzitutto selezionarla. L'AID è la stessa già illustrata nel codice al Paragrafo 6.1.

5) L'invocazione del metodo dumpHeap (cfr Paragrafo 5.4) deve specificare nei dati passati l'AID del *contesto* di cui si vuole vedere gli *Oggetti*. Gli 8 bytes che si ottengono in ritorno sono la codifica delle informazioni riguardanti l'*Oggetto* creato. Nell'ordine di vede che:
   - **40** è il codice che sta per "Nome Oggetto"
   - **00,01** è il nome dell'oggetto creato, uno short (2 bytes) deciso da un contatore il cui valore è inizialmente 1 (cfr Paragrafo 5.3.1).
   - **41** è il codice di "Contatore Oggetto"
   - **00,02** è il valore del contatore dell'*Oggetto*, sta su uno short e vale due perché ci sono due riferimenti che puntano a tale oggetto.
   - **42** è il codice di "Dimensione Oggetto"
   - **02** è il numero di bytes che tale *Oggetto* occupa. Questa informazione è memorizzata in un byte nella struttura dati del modello.
6) L'invocazione del metodo dumpTabRif (cfr Paragrafo 5.4) deve specificare nei dati passati l'AID del *contesto* di cui si vuole vedere la *Tabella dei Riferimenti*. Dal momento che i *Riferimenti* sono due, la sequenza di 16 bytes che si ottiene è così fatta:
   - **30** sta per "codice nome riferimento"
   - Seguono i 2 bytes del *codice short* che l'utente ha deciso per ogni *Riferimento* (cfr Paragrafo 5.2.3.2). Si vedono **B7,16** per il primo (`ALIAS_ID`) e **C7,65** per il secondo (`VETT_ID`).
   - Il **40** fa capire che quello che segue dopo è un "Nome Oggetto", qui interpretato come quello a cui punta il *Riferimento*, ovvero lo **00,01** di cui al punto 5) per entrambi.
   - L'ultimo valore, anch'esso uguale per entrambi i *Riferimenti*, è il numero di istanza in cui è stato creato il *Riferimento*. Tale codice è su un byte, vale **01**, ed è preceduto dal codice **31**.
7) L'ultimo metodo invocato in questa sessione di CAD è l'ActionQueue, destinato a stampare a video tutto il *fileLog* della *gcApplet*. La serie di 90 byte è in ordine cronologico e registra, nell'ordine, le seguenti *Azioni*:
   - L'installazione dell'applet, con codice **76**, è seguita da **7a**, che sta per "nome applet", e poi dall'AID del package (sono 7 bytes).

- E' stato creato un *Oggetto*, e **73** è il codice della creazione *Oggetto*, cui deve seguire **7a** e poi l'AID per capire in quale *contesto* è stato creato (in più contesti ci potrebbero essere *Oggetti* con nomi uguali[*]). Il nome dell'*Oggetto* è su 2 bytes ed è preceduto, come è stato visto, da **40** (vale **00,01**), poi **41** e il valore del contatore (vale **00,01**), poi la dimensione (vale **02**) preceduta da **42**.
- L'azione che segue è la creazione di un *Riferimento* (codice 70). L'applet AID va ripetuto poiché, gestendo più applet, potrebbe inserirsi la creazione di *Oggetti* o *Riferimenti* appartenenti a *contesti* diversi. Il nome del *Riferimento* (`byte[] vett`) è **c7,65**, punta all'*Oggetto* di nome **00,01** e è stato creato nell'istanza **01**. I codici sono gli stessi della dumpHeap e dumpTabRif.
- Subito dopo e con le stesse modalità del precedente, è creato il *Riferimento* **b7,16** (`byte[] alias`), che punta inizialmente a **null**, infatti il codice "Nome Oggetto" è seguito da **00,00**.
- Il riferimento `alias` viene, subito dopo, assegnato dall'*Oggetto* puntato da `vett`, e quindi l'azione che si registra successivamente è una modifica del contatore dell'*Oggetto* puntato. Il codice della modifica oggetto è 75, succeduta dall'AID e poi dal nome *Oggetto* (cioè **00,01**) e valore contatore che, incrementato, sale a **00,02**[†].
- L'assegnamento produce anche una modifica del *Riferimento* in termini di *Oggetto* riferito. Tale modifica ha codice **72** a cui segue il solito AID, il nome del *Riferimento* (**b7,16**) e il nome del nuovo *Oggetto* riferito (**00,01**)[‡].

8) Alla fine la sessione di CAD si chiude con il comando powerdown che termina la comunicazione simulando la cessazione dell'alimentazione della carta.

La Tabella che segue riporta tutti i codici utilizzati dalla dumpHeap, dumpTabRif e actionQueue nella descrizione, rispettivamente, degli *Oggetti*, dei *Riferimenti* e della *codaAzioni*.

---

[*] Anzi, poiché tale nome deriva da un contatore che parte da uno, sicuramente esisteranno in più contesti oggetti con nomi uguali.

[†] Il valore relativo alla dimensione non può essere cambiato poiché deciso all'atto della allocazione.

[‡] Il valore relativo all'istanza, infatti, non è passibile di modifica a run-time poiché stabilito all'atto della creazione

Codice HEX	Significato
70	Inserimento nuovo Riferimento
71	Cancellazione Riferimento
72	Modifica Riferimento
73	Inserimento nuovo Oggetto
74	Cancellazione Oggetto
75	Modifica Oggetto
76	Installazione nuovo Package
7a	Package AID
30	Nome Riferimento
31	Istanza Riferimento
40	Nome Oggetto
41	Contatore Oggetto
42	Dimensione Oggetto

Tabella 6. 1 Codici di interpretazione dell'output

7. Conclusioni

Il sistema Java Card è un sistema embedded con forti limitazioni di risorse su cui è possibile una coesistenza di più applicazioni.

L'introduzione di un algoritmo di Garbage Collection nella Virtual Machine delle Java Card, solleverebbe il programmatore dalla preoccupazione di implementare routine apposite per la gestione della memoria.

In questa tesi è stato sviluppato un Garbage Collector per Java Card basato sul Reference Counting. Il prototipo sviluppato implementa la Garbage Collection installandosi sulla carta come una normale applet, e instaurando un meccanismo di comunicazione che evita l'isolamento del firewall di sistema mediante la condivisione di uno *Shared Interface Object (SIO)*.

Questa scelta implementativa si è resa necessaria a causa dell'impossibilità di operare direttamente sulla Virtual Machine, ed ha introdotto vincoli sul codice delle Applet per il corretto utilizzo del garbage Collector.

Le regole di chiamata ai metodi nascono dall'idea di introdurre una sorta di *write-barrier* a protezione di tutta la memoria utente, e, se rispettate, garantiscono che la **gcApplet** si creerà un modello affidabile e veritiero di tale memoria.

La scelta del Garbage Collector è ricaduta su uno di tipo non conservativo, incrementale e di Reference Counting con informazioni di tipo Tracing.

La scelta è derivata dalla necessità di un collector di tipo incrementale che non appesantisca troppo il tempo di computazione, e dal fatto di rendere più veloci ed efficienti le collezioni migliorando il *reference counting* con informazioni circa la struttura dei riferimenti e le dipendenze di oggetti riferiti tipiche di algoritmi di tipo *tracing*. La struttura dati che conferisce questa variante è la *Tabella dei Riferimenti*.

Possibili sviluppi futuri sono l'implementazione della gcApplet nella Virtual Machine. Inoltre la Tabella dei Riferimenti rende flessibile il Garbage Collector. Sarebbero infatti necessarie poche modifiche per implementare un algoritmo di tipo tracing, ad esempio generazionale.

8. Bibliografia

[GCFAQ]

iecc.com, GCFAQ

[JVM]

Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification Second Edition, Addison Wesley*

[IBM]

ibm.com developerWorks, Java Technology, Brian Goetz, *Java theory and practice*

[IBMGC]

ibm.com Sensible Sanitation – Sam Borman, *Understanding the IBM Java Garbage Collector* (http://www-106.ibm.com/developerworks/ibm/library/i-garbage1/)

[JavaWorld]

Javaworld.com, Jeff Friezen, *Trash Talk*

[Bull]

Produttore di smart card (www.bull.com)

[Chen]

Zhiqun Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide, Addison Wesley 2000*

(http://java.sun.com/developer/Books/consumerproducts/javacard/)

[Axalto]

Schlumberger, un produttore di smart card (www.cyberflex.com)

[Gemplus]

Produttore di smart card (www.gemplus.com)

[GemplusOS]

Gemplus Software research, Laurent Lagosanto, *Next Generation Embedded Java Operating Systems for Smart Cards*

[JCForum]

The Java Card Forum (www.javacardforum.org)

[JCspec2.2.1]

Java Card 2.2.1 platform specification (http://java.sun.com/products/javacard/specs.html)

[USENIX]

Techical program paper – smartcard, Marcus Oestreicher and Ksheerabdhi Krishna *Object Lifetimes in Java Card* (www.usenix.org)

[Hot Spot]

Java Hot Spot Technology, white paper (http://java.sun.com/products/hotspot/index.html)

[JCKit2.2.1]

Java Card Development Kit for the Java Card (http://java.sun.com/products/javacard/dev_kit.html)

[devX]

Tony Printezis, *Garbage Collection in Java HotSpot Virtual Machine* (www.devx.com/Java/Article/21977)